DJANGO

THE EASY WAY

Samuli Natri

Django - The Easy Way

A step-by-step guide on building Django websites

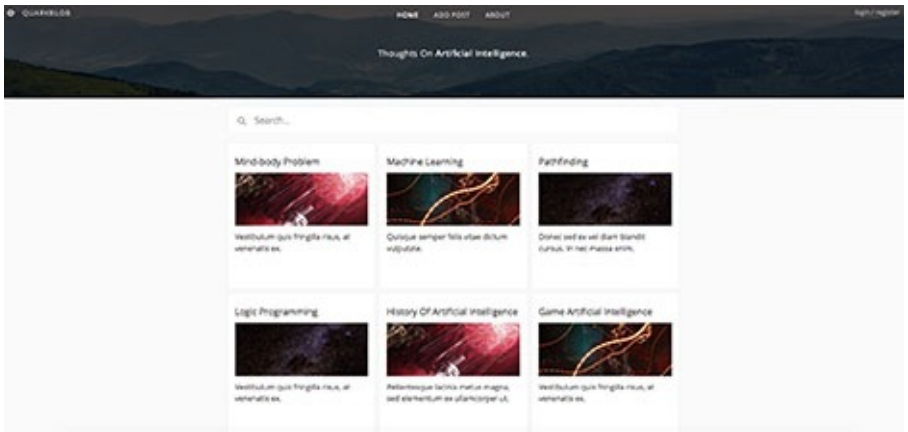
Samuli Natri

Table of Contents

1. Foreword

About This Book

This book is a **practical** guide on how to build **Django** applications. We will build a *blog* from start to finnish. Each new *feature* that we add introduces a new concept.



We cover topics from **basic concepts** like *Models* and *Views* all the way to **test driven development** and **deployment**. By the end of this book you will have deployed a *responsive*, *cached* website to a *production* server and learned how to **update** it.

The focus is on **Django topics**. This is **not** a *Python*, *HTML*, *CSS* or *JavaScript* book. Those will be covered minimally.

Code Blocks

Sometimes long lines have extra **backslashes** ("\") that have been added *automatically*. Make sure not to include those.

New **inserts** and **changes** in code are marked with **bold** text:

```
INSTALLED_APPS = [
...
    'django.contrib.staticfiles',
    'base',
    'blog',
]
```

Editors, Code Hosting & Operating System

I would recommend starting with an **editor** that is **easy** to get into like "**Atom**" (https://atom.io/) so you don't have to fight the tool while learning **Django**. Check also **PyCharm Community Edition** (https://www.jetbrains.com/pycharm/). It's a **free** tool for *Python* development.

My alltime favourite is **Emacs** (https://www.gnu.org/software/emacs/) but it requires some work for it to be *comfortable* to use.

With **server** operations I usually use "**Vim**" (https://www.vim.org/) but **Nano** (https://www.nano-editor.org/) might be more suitable for beginners. You can install it like this:

```
sudo apt-get update
sudo apt-get install nano
```

You can use "**BitBucket**" (https://bitbucket.org/) to host your code repository for **free**. Checkout also *GitHub* (https://github.com/) and *GitLab* (https://about.gitlab.com/).

I'm using "**macOS**" but the development process is *pretty* much the same with all operating systems. The differences are *mainly* manifesting in terminal operations. For example in **macOS** you would activate *virtual environment* like this "`source venv/bin/activate`" but in **Windows** you run the *.bat* executable "`venv\Scripts\activate.bat`".

About The Author

Samuli Natri has been doing software development since the 90's. He attended **Helsinki University Of Technology** (Computer Science) and **Helsinki University** (Social Sciences).

2. New Project

Install **Python 3**: https://www.python.org/downloads/.

I'm going to call this project "**QuarkBlob**". Let's **organize** the files like this:

```
quarkblob
├── project
│   ├── base
│   ├── blog
│   ├── feedback
│   ├── manage.py
│   ├── project
│   └── search
├── venv
└── assets (optional)
```

"**project**" folder contains the *Django web application*. These are the files we will *deploy* to the production environment in the *Deployment* chapter.

"**venv**" is the **virtual environment** folder.

> "**VIRTUAL ENVIRONMENTS**" | **Django** is like any other Python package and will be installed inside the **venv** folder. Also the **Python** executable will be installed in the same folder. This way you can isolate Python and package versions from the system and other projects. We don't copy this folder to the *production* environment. Instead we install the dependencies (Django and other packets) using **pip** and *requirements.txt* file. Read more about *venv*: https://docs.python.org/3/tutorial/venv.html.

“**assets**” is an optional folder where you can store for example **mockups** or other related **documents**.

Inside the Django **project** folder we have “**app**” folders and other relevant files.

> “**APPS**” are Python *packages* that live inside the project folder and provides **features**. You can *reuse* apps in multiple projects. Apps contain *views*, *models* and *templates* among many things. You usually add new apps to the **INSTALLED_APPS** list in the *settings.py* file.

- “**base**” app is where we store common reusable items like the **base layout**.
- “**blog**” app provides the *blogging* functionality for the site.
- We also create apps for “**feedback**” and “**search**” functionality.
- Inside the project folder we have *another* “**project**” folder (*quarkblob/project/project/*). This is also considered an *app*. In here we put files like “**settings.py**” that contains *site wide configuration* data.

Create the **project folder** somewhere in your system:

```
mkdir quarkblob && cd quarkblob
```

Create a **virtual environment** and activate it:

```
python3 -m venv venv && source venv/bin/activate
```

You might have multiple Python *versions* installed in your system. That’s why we specify the version with “**python3**”.

In **Windows** *activate* the virtual environment by running "**`venv\Scripts\activate.bat`**".

Install **Django package** and create a new **Django project**:

```
pip install django && django-admin startproject project && cd
project
```

> "**PIP**" is a **Python package manager**. Read more: https://pip.pypa.io/en/stable/. You can upgrade it with "**pip install –upgrade pip**".

> "**DJANGO-ADMIN**" is a **command line tool** for administrative tasks. After we have installed the Django project with it we will use "**manage.py**" from the project folder. It does mostly the same thing as *django-admin* but it's easier to use when working with a specific project.

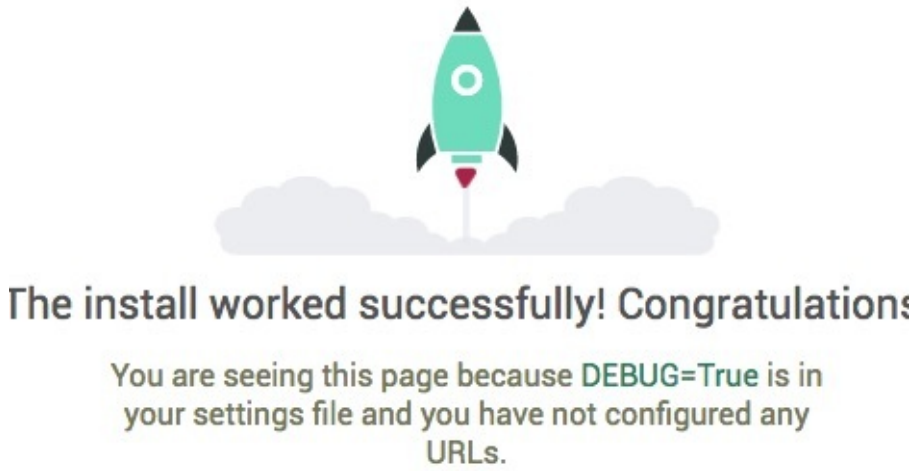
Run the **development server**. We can now use "**python**" instead of "**python3**" when the virtual environment is *activated*:

```
python manage.py runserver
```

> "**RUNSERVER**" starts a **development web server**. Don't use this with **live** sites. In the *Deployment* chapter we will use **Nginx** and **Gunicorn** to serve the pages from the *production* server.

You can ignore the notice about "**unapplied migrations**" for now.

Visit “**http://127.0.0.1:8000/**” and you should see the welcome screen:



3. Base App, PyCharm And Homepage

Let's create a **base** app to hold common reusable items like the main **HTML skeleton**.

PyCharm For Django

Open the project folder in your **editor**. In **PyCharm** you can do this:

- Go to "**File > Open**" and open "**quarkblob/project/**".
- Go to "**PyCharm > Preferences**" and search for "**interpreter**".
- Hit the grey **cog** and select "**Add Local**".
- Select the python3 *executable* ("**quarkblob/venv/bin/python3**") as the "**Existing environment**".

Base App

Open another **tab** in the **terminal** (let the development server run on the other). Make sure to activate the virtual environment in the new tab:

```
source ../venv/bin/activate
```

Create the **base** app:

```
python manage.py startapp base
```

Edit "**settings.py**" in *quarkblob/project/* and add **base** app to the **INSTALLED_APPS** list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
```

```
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'base',
]
```

> “**PYCHARM TIP**” | You can **find files** easily by using the “**Search Everywhere**” tool. Press “**Shift**” two times to access it.

Base Template

Create a file “**base.html**” (and the directories) in *quarkblob/project/base/**templates/base/***.

> “**PYCHARM TIP**” | Select the “**quarkblob/project/base**” folder and **create a new file** with “**Ctrl + Alt + N, F**”. Write “**templates/base/base.html**” in the field and it will create the *parent directories* as well.

This is where the “**base.html**” template should live:

```
base
├── templates
│   └── base
│       └── base.html  # here
project
```

> “**TEMPLATES FOLDER**” | It’s preferable to organize templates in subdirectories. And furthermore it’s a *convention* to use the app name when creating the subfolder. That’s why we have **base** folder

inside the **templates** folder. Django will search for these templates automatically. For example we can now refer to this template simply using **base/base.html**. Read more about *templates*: https://docs.djangoproject.com/en/dev/topics/templates/#django.template.b

Add these lines in **base.html**:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>QuarkBlob</title>
</head>
<body>
{% block content %}{% endblock %}
</body>
</html>
```

"**TEMPLATE FILES**" | Django use templates to **generate HTML** dynamically. Template is a combination of **static** and **dynamic** content. The **content** block in *base.html* will contain different *dynamic* content depending on the page the user is visiting. Rest of the elements are currently *static*: they will stay the same for all pages.

Home URL

Open "**urls.py**" in *quarkblob/project/project/* and make the following changes:

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('', include('base.urls'))
]
```

> "**URLPATTERNS**" | Django runs through the **urlpatterns** list. When the pattern matches the requested url, it stops. So if we visit **/admin/** url, then Django will match the request and stop at the **'admin/'** path.

> "**FORWARD SLASH**" | Django will by default *redirect* to the same url with *slash appended* if the path doesn't end in a slash and doesn't match any patterns. So visiting **/admin** will take you to **/admin/**. If we would allow access to resources in both */admin* and */admin/* urls, then search engines would store multiple entries for the same resource. Read more: https://docs.djangoproject.com/en/dev/ref/settings/#append-slash

> "**INCLUDE**" | We can **include** urls from other *modules* but we could as well put **all** urls in one file. In this book I organize some of the urls in separate files and put some general urls in the main urls file (*quarkblob/project/project/**urls.py***).

Create "**urls.py**" in *quarkblob/project/base/* and add these lines:

```
from django.urls import path

from base import views

urlpatterns = [
```

```
    path('', views.home, name='home'),
]
```

In here we created a **urls.py** file for the base app to store url pattern for the **homepage**.

> "**VIEW CALL**" | Once the pattern matches the *root* of the site, Django will call **home** *view* from the *views* module in *base* app. In this case that will be a Python function called **home** but it could also be a *class based* view. The view gets several **arguments** like the *HttpRequest* object.

> "**HTTPREQUEST**" | When page is requested, Django creates an **HttpRequest** object that contains data about that request. This includes **attributes** like *HttpRequest.path* that contains the full path to the requested page (like **/admin/**) or *HttpRequest.META* that contains HTTP header data (like visitor **IP address**).

> "**URL REVERSING**" | "**name='home'**" allows us to get this url using its name "**home**" in different contexts. Using "**{% url 'home' %}**" in templates prints out the path to the homepage. This also allows us to get the *translated* version of the path. In Python code you can use the "**reverse()**" function. We will use *both* of these technics later.

Home View

Edit "**views.py**" in *quarkblob/project/base/* and add these lines:

```python
from django.shortcuts import render

def home(request):
    return render(request, 'base/home.html')
```

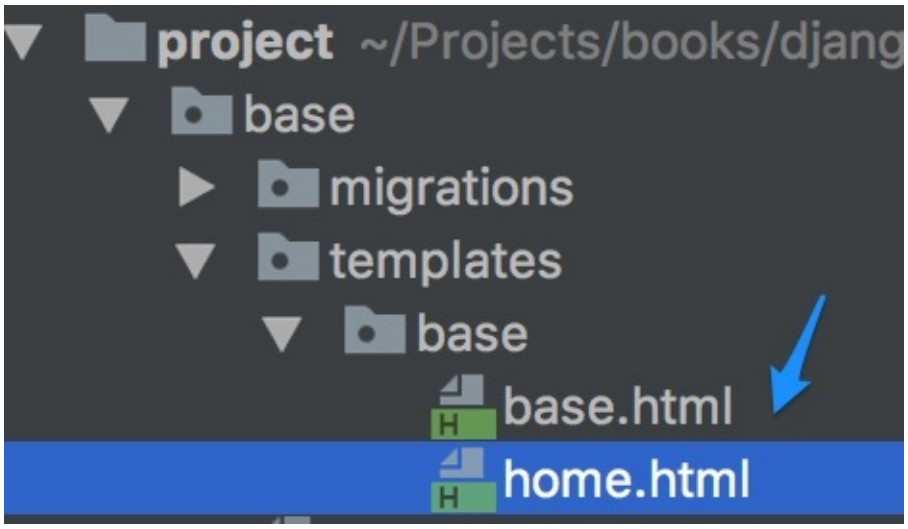
> "**VIEWS**" | "**home**" function is a **view**. View function takes a **request** as an argument and returns a **response**. "**request**" is the **HttpRequest** object we mentioned earlier. Django creates it automatically. The "**response**" can be a lot of things like **HTML contents** for the web page, *redirect* or an *image*. The user is responsible for creating the **HttpResponse** object. In this case we use the *render* function that returns the *HttpResponse* object. To put it simply: we **render** the *home.html* template.

Home Template & Inheritance

Create "**home.html**" in *quarkblob/project/base/templates/base/* and add these lines:

```
{% extends 'base/base.html' %}
{% block content %}<h1>Homepage</h1>{% endblock %}
```

Home template should now be in here:



> "**TEMPLATE INHERITANCE**" **|** We use **extends** tag to indicate that this template *extends* a **parent** template. It loads the *parent* template ("**base.html**") and replace the **content** block inside it with the **content** block from the *child* template ("**home.html**").

Refresh browser and you should see the homepage with just a black "**Homepage**" text.

> "**SERVER ERRORS**" **|** Check the server tab for any errors. You might occasionally need to shutdown the **development server** with "**Ctrl + C**" and run it again.

4. Version Control With GIT (Optional)

This would be a good time to start **version controlling** your project. I will use **GIT** in this book.

> "**GIT**" version control system **tracks changes** in computer files. Using it is **optional** but highly recommended if you are serious about software development. You don't need to know a lot about it to get started. In this chapter I will cover briefly on how to use it with Django. We will also use it in the *Deployment* chapter to move files to the *production* server.

Visit https://git-scm.com/download and **install** it to your operating system.

.gitignore file

Create **.gitignore** file in the *root* of the site: *quarkblob/*

```
# DJANGO
__pycache__/
*.py[cod]
db.sqlite3
settings.py
# OTHER
node_modules
.DS_Store
.idea
media
```

> "**.GITIGNORE**" contains patterns for files we don't want to include in the **repository**. You want to ignore at least **cache**, **media** files, **database** and **settings.py**. Note that files like **.gitignore** might be hidden in your system. Here are more .gitignore options: https://github.com/jpadilla/django-project-template/blob/master/.gitignore.

First Commit

Initialize the **repository** and make a first **commit**:

```
git init && git add . && git commit -m "initial"
```

You can use these to see **status** and **log**:

```
git status && git log
```

Use **q** to exit log.

> "**COMMITS**" are like **snaphots** of the project files at specific moments. By manipulating these commits you can for example *go back* in the project change history.

Bitbucket & SSH Key

Let's use **Bitbucket** to store our repository in a **remote** location. It's **free**.

- Go to https://bitbucket.org/ and **create an account**.

- Setup an **SSH** key: https://confluence.atlassian.com/bitbucket/set-up-an-ssh-key-728138079.html

You can **generate** the key in **macOS** like this (just hit return for all questions):

```
ssh-keygen
```

Copy the contents of the generated file (**~/.ssh/id_rsa.pub**).

This will print the contents to the screen:

```
cat ~/.ssh/id_rsa.pub
```

- Go to the account **Settings** page https://bitbucket.org/account and click **SSH Keys**.
- Hit **Add key**.
- **Paste** your key there and **save**.

New Repository

Go to https://bitbucket.org/repo/create, fill in the name (**quarkblob-book**) and hit **create repository**.

Go to your project root in the command line (**quarkblob/project/**) and use these commands to setup the remote:

```
git remote add origin git@bitbucket.org:YOUR_ACCOUNT/quarkblob-
book.git
git push -u origin master
```

Refresh the project webpage in Bitbucket and you should see the project files.

Push

Let's test on how to **add**, **push** and **delete** an app.

```
python manage.py startapp mytest
git status
git add .
git commit -m "Add mytest app"
git push
```

Refresh the project webpage in Bitbucket and you should see the **mytest** app folder.

Delete it like this:

```
git rm -fr mytest
git status
git commit -m "Delete mytest app"
git push
```

You can find all **commits** in here https://bitbucket.org/YOUR_ACCOUNT/quarkblob-book/commits/all:

Author	Commit	Message
Samuli Natri	157e440	Delete mytest app
Samuli Natri	9a8d27d	Add mytest app
Samuli Natri	3629e9d	initial

This should get you started. Try to make *many* **small** commits, not a few *giant* ones.

5. CSS Styling, Gulp, SASS

The website will look quite *ugly* if we don't do any **styling**. This means applying **CSS** styles to elements so we can do things like change *color*, *font size* and *arrange* layout.

I will be using **SASS** preprocessor to ease out with generating that **CSS**. **Gulp** is used to compile *SASS* files. You don't have to use *preprocessors* but it will make your life easier.

CSS

Edit "**base.html**" in *quarkblob/project/base/templates/base/* and add these lines:

```
<!doctype html>
{% load static i18n %}
<html>
<head>
    <meta charset="utf-8">
    <title>QuarkBlob</title>
    <link href="{% static 'base/vendor/normalize/normalize.css'
%}" rel="styl\
esheet">
    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.1.0/\
css/all.css"  integrity="sha384-
lKuwvrZot6UHsBSfcMvOkWwlCMgc0TaWr+30HWe3a4lta\
BwTZhyTEggF5tJv8tbt" crossorigin="anonymous">
    <link href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400,600\
" rel="stylesheet">
    <link rel="stylesheet" href="{% static 'base/css/style.css'
%}">
</head>
```

“**{% load static i18n %}**” allows us to use “**{% static %}**” tag to load **static** assets like **CSS** and **JS**. If you look at the homepage source code, you can now see that the link element for normalize points to “**/static/base/vendor/normalize/normalize.css**”.

Sometimes you might want to serve the static files from another server. Now we can just change the static variable in the **settings.py** file so the file could be served from urls like this “**https://static.quarkblob.com/…normalize.css**”.

“**normalize.css**” renders elements more **consistently** across all browsers. Download it from here “https://necolas.github.io/normalize.css/” and put it in here “*base/static/base/vendor/normalize/**normalize.css***”.

> “**FONTAWESOME.COM**” offers free and paid icons and logos. We will use the Fontawesome library for the site **logo** and **search** icon. Copy *latest* link from here: https://fontawesome.com/how-to-use/on-the-web/setup/getting-started.

> “**GOOGLE FONTS**” provides a selection of **free** fonts: https://fonts.google.com/. We will be using **Open Sans** as the main font for the site.

Create “**style.css**” file in *quarkblob/project/base/static/base/css/* and add these lines:

```
h1 {
   font-size: 100px;
}
```

Restart the development server.

You should now see the **Homepage** text in much larger size.

> "**CHANGES ARE NOT SHOWING?**" | If nothing seems to change and you are using Chrome browser, open the **Developer tools** from *View* > *Developer* > *Developer tools*. You might also need to select **Network** > **Disable cache**. Other browsers like **Firefox** and **Safari** have similar *inspector* tools.

Vanilla CSS

You don't have to use **Gulp** or **SASS** to continue with this book. You can put all your styling in the **style.css** file and use *vanilla CSS*.

Here is an example of *vanilla CSS* syntax:

```
.menu {
  list-style: none;
  text-align: center;
  padding-left: 0;
}
.menu-li {
  display: inline-block;
}
.menu-li-a {
  display: inline-block;
  color: #ccc;
}
.menu-li-a--active, .menu-li-a:hover {
  color: #fff;
}
```

Here is the same thing with *SASS*:

```
.menu {
    list-style: none;
```

```
    text-align: center;
    padding-left: 0;
  &-li {
    display: inline-block;
    &-a {
      display: inline-block;
      color: #ccc;
      &--active, &:hover {
        color: #fff;
      }
    }
  }
}
```

At the end the SASS example above will be **compiled** into *vanilla CSS* anyway.

> "**EXPLICIT TARGETING**" | I'm creating CSS that targets elements as directly as possible. So instead of doing "**.menu > li**", I will do "**.menu-li**". It's **faster** to target **id** or **class** instead of using a *child* selector. Also we are dealing with *lower specifity*: https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity. Also don't use **!important** rule unless you have to.

> I'm using a "**CSS SYNTAX**" *similar* to the **Block Element Modifier** methodology: http://getbem.com/. Why **BEM?** Read more: https://blog.decaf.de/2015/06/24/why-bem-in-a-nutshell/.

Gulp

> "**GULP**" is a useful **automation tool**. We will use it to watch and **compile SASS** files but you can do much more with it like refresh

> the browser automatically when files change.

Install **Node.js**: https://nodejs.org/en/.

> With "**NODE.JS**" you can run **JavaScript** code outside the browser. Task-runners like **Gulp** and **Grunt** are built on it.

Open a new terminal **tab**, go to your Django project root (*quarkblob/project/*) and install **Gulp**:

```
sudo npm install gulp -g
sudo npm install gulp --save-dev
sudo npm install gulp-sass --save-dev
```

In **Windows** leave "*sudo*" out.

> "**NPM**" is a **package manager** for *JavaScript*.

I'm not going in detail on *how to use Gulp* but here is a simple example on how to **watch** the "**style.scss**" file and compile it to "**style.css**".

Create "**gulpfile.js**" file in *quarkblob/project/*:

```
var gulp = require('gulp');
var sass = require('gulp-sass');

gulp.task('sass', function(){
    return gulp.src('base/static/base/css/style.scss')
        .pipe(sass())
        .pipe(gulp.dest('base/static/base/css'))
```

```
});

gulp.task('default', function(){
   gulp.watch('base/static/base/css/*.scss', ['sass']);
});
```

This will use *base/static/base/css/**style.scss*** as source and compile it to plain CSS in *base/static/base/css/**style.css***.

> "**GUI APPLICATIONS**" | There are some *desktop applications* that you can use to **compile SASS**. Check out https://prepros.io/, https://codekitapp.com/ and http://koala-app.com/.

SASS

> "**SASS**" is a **style sheet language** that helps you write *less* and more *organized* style files.

Create "**style.scss**" file in *quarkblob/project/base/static/base/css/* and fill it with these lines:

```
h1 {
   font-size: 100px;
   color: red;
}
```

Open new *tab*, use the "**gulp**" command and keep it running.

```
gulp
```

Make some changes to the "**style.scss**" file in *quarkblob/project/base/static/base/css/* and *save* it to start the *compiling* process.

You should see something like this in the *terminal*:

```
[17:34:17] Starting 'default'...
[17:34:17] Finished 'default' after 11 ms
[17:34:19] Starting 'sass'...
[17:34:20] Finished 'sass' after 42 ms
```

"**Homepage**" text should now have a *red* color.

We are going to separate styling into multiple files. For example, all styling related to *tags* goes in **tags.scss** and all styling related to *pagination* goes in **pagination.scss**.

Finally all styling will be compiled into one **style.css** file:

6. Header And Footer

Let's add **header**, **footer** and **main** *containers*.

Markup

Open "**base.html**" in *quarkblob/project/base/templates/base/* and replace the **body** element with these lines:

```
<body>
<div class="header">
    <a class="header-brand" href="{% url 'home' %}">
        <i class="header-brand-logo fas fa-futbol"></i>
        <span class="header-brand-name">QuarkBlob</span>
    </a>
    <div class="header-slogan">Thoughts On <strong>Artificial
Intelligence</s\
```

```
trong>.</div>
</div>
```

```
<div class="main">
    {% block content %}{% endblock %}
```

```
</div>
```

```
<div class="footer"></div>
<div class="footer-bottom">
    @ {% now "Y" %} QuarkBlob. All rights reserved.



</div>
</body>
```

“**URL TAGS**” | “**{% url ‘home’ %}**” tag generates a link to the homepage according to the **home** url pattern. It’s more flexible to use **url** tag instead of **hard-coding** paths like this: “**href=”/about/””**”. This allows you to **change** the path in one place and the new path will be used everywhere you use the url tag. This also allows us to **translate** paths easily as we will do later.

“**LOGO**” | You can add *Fontawesome* assets using the **i** element and specific *class names*: “**<i class=”fas fa-futbol”></i>**”. We will use this as the **logo**: https://fontawesome.com/icons/futbol?style=regular&from=io.

“**NOW TAG**” | “**{% now “Y” %}**” prints out the **current year** so we don’t have to *hard-code* it. See more built-in tags: https://docs.djangoproject.com/en/dev/ref/templates/builtins/#built-in-tag-reference

Styling

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and replace the content with these lines:

```
@import "../vendor/normalize/normalize.css";
@import "main";
@import "header";
@import "footer";
@import "layout";
```

Create “**main.scss**” in *quarkblob/project/base/static/base/css/*:

```
*, *:after, *:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}


body {
  font-family: "Open Sans", sans-serif;
  background-color: #FAFAFA;
}


p {
  line-height: 1.5em;
}
```

```
a {
  text-decoration: none;
  color: #333;
}


h1 {
  font-size: 30px;
  font-weight: 300;
  margin-top: 0;
}


ul {
  padding-top: 0;
}


.clearfix {
  clear: both;
}
```

This file will contain all **general** styling. You could do more granual arrangement like separate *typography* in its own file.

> "**border-box**" value for **box-sizing** *property* makes it easier to size elements. It makes *padding* and *border* part of the width. So now if you specify width as "**200px**" for some element, the width will be exactly that no matter what *padding* or *border* you give to the element.

Create “**layout.scss**” in *quarkblob/project/base/static/base/css/*:

```
.main {
  margin: 1em auto;
  width: 60em;
  min-height: 500px;
  padding: 2em;
  background-color: #fff;
  border: 1px solid #E8E8E8;
  border-radius: 3px;
}
```

Create “**header.scss**” in *quarkblob/project/base/static/base/css/*:

```
.header {
  height: 200px;
  width: 100%;
  background: url('../images/header-bg.png') no-repeat;
  &-brand {
    position: absolute;
    text-transform: uppercase;
    letter-spacing: 2px;
    font-size: 14px;
  }
  &-brand-logo {
    padding: 1em 0.7em 1em 1em;
    color: #fff;
  }
  &-brand-name {
    color: #ccc;
  }
  &-slogan {
    font-size: 18px;
    text-align: center;
    color: #fff;
    padding-top: 5em;
    font-weight: 300;
    letter-spacing: 1px;
```

```
  }
}
```

Create “**footer.scss**” in *quarkblob/project/base/static/base/css/*:

```
.footer {
  text-align: center;
  height: 200px;
  line-height: 200px;
  width: 100%;
  background: url('../images/header-bg.png') no-repeat;
}


.footer-bottom {
  height: 50px;
  text-align: center;
  line-height: 50px;
  font-size: 0.9em;
  background-color: #414141;
  color: #aaa;
  &-a {
    color: #aaa;
  }
}
```

> “**IMAGES**” | Create a folder *quarkblob/project/base/static/base/**images/*** and **download** the images from here: https://github.com/SamuliNatri/DjangoTheEasyWay.

You should now see something like this:

"**ERRORS**" **|** Make sure to check the tabs for **Gulp** and **Django development server** for any errors.

7. Breakpoints And Main Menu

Let's specify **breakpoints** so we can have a **responsive** design. I will use **@media** rule and **mixins** to apply different styles for different screen sizes.

Mixins

> "**MIXINS**" allows you to **reuse** CSS declaration blocks.

Create "**breakpoints.scss**" in *quarkblob/project/base/static/base/css/*:

```
@mixin for-phone-only {
  @media (max-width: 599px) { @content; }
}

@mixin for-tablet-portrait-up {
  @media (min-width: 600px) { @content; }
}

@mixin for-tablet-landscape-up {
  @media (min-width: 900px) { @content; }
}

@mixin for-999-up {
  @media (min-width: 999px) { @content; }
}

@mixin for-desktop-up {
  @media (min-width: 1200px) { @content; }
}

@mixin for-desktop-menu-up {
  @media (min-width: 1400px) { @content; }
```

```
}

@mixin for-big-desktop-up {
  @media (min-width: 1800px) { @content; }
}
```

I’m using *roughly* the widths for common portable devices and desktop screen sizes. You can add more of them if needed.

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/*:

Import **breakpoints**:

```
@import "../vendor/normalize/normalize.css";
@import "breakpoints";
@import "main";
```

Responsive Layout

Edit “**layout.scss**” in *quarkblob/project/base/static/base/css/*.

Now we can use **@include** *directive* to apply mixins:

```
.main {
  margin: 1em auto;
  width: 60em;
  min-height: 500px;
  padding: 2em;
  background-color: #fff;
  border: 1px solid #E8E8E8;
  border-radius: 3px;
  @include for-999-up {
    width: 60em;
  }
}
```

If you are not using **SASS** and **mixins**, you can do this:

```
.main {
  margin: 1em auto;
  width: 60em;
```

```
  min-height: 500px;
  padding: 2em;
  background-color: #fff;
  border: 1px solid #E8E8E8;
  border-radius: 3px;
}

@media (min-width: 999px) {
  .main {
    width: 60em;
  }
}
```

So now the **main** container will take the whole width of the browser window until the browser window size grows to **999px**:

Edit "**header.scss**" in *quarkblob/project/base/static/base/css/*.

Add this include for **slogan**:

```
&-slogan {
    font-size: 18px;
    text-align: center;
    color: #fff;
    padding-top: 5em;
    font-weight: 300;
    letter-spacing: 1px;
    @include for-desktop-up {
      padding-top: 2.5em;
    }
  }
```

This positions the slogan correctly whether the top menu is visible or not.

Main Menu

Edit “**base.html**” in *quarkblob/project/base/templates/base/*.

Add the **menu** markup between the **header-brand** and **header-slogan** elements:

```
<div class="header">
    <a class="header-brand" href="{% url 'home' %}">
        <i class="header-brand-logo fas fa-futbol"></i>
        <span class="header-brand-name">QuarkBlob</span>
    </a>
    <ul class="menu">
        <li class="menu-li">
            <a class="menu-li-a {% url 'home' as home_url %} {%
if request.ge\
t_full_path == home_url %} menu-li-a--active{% endif %}" href="
{% url 'home' \
%}">Home</a>
        </li>
        <li class="menu-li">
            <a class="menu-li-a" href="">About</a>
        </li>
    </ul>
    <div class="header-slogan">Thoughts On <strong>Artificial
Intelligence</s\
trong>.</div>
</div>
```

“**{% url ‘home’ as home_url %}**” stores the home url in a variable called **home_url**.

“**request.get_full_path**” returns the **path** and possibly *query string* like “**q=search**”.

“**{% if request.get_full_path == home_url %} menu-li-a–active {% endif %}**” adds “**menu-li-a–active**” class to *active* items.

Create “**menu.scss**” in *quarkblob/project/base/static/base/css/*:

```
.menu {
    list-style: none;
    text-align: center;
    padding-left: 0;
    color: #fff;
    margin: 0;
    display: none;
    @include for-desktop-up {
      display: inherit;
    }
  &-li {
    display: inline-block;
    &-a {
      display: inline-block;
      color: #ccc;
      height: 50px;
      line-height: 50px;
      padding: 0 1em;
      margin: 0;
      font-size: 14px;
      text-transform: uppercase;
      letter-spacing: 1px;
      &--active, &:hover {
        color: #fff;
      }
    }
  }
}
```

The following SASS is used to **hide** the main menu until the browser window size grows big enough:

```
 ...
 display: none;
 @include for-desktop-up {
      display: inherit;
    }
 ...
```

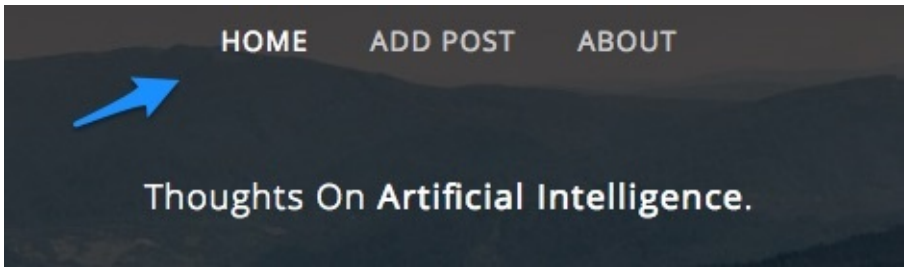
"**display: inherit**" sets the display the same as its parent.

Later we will build a **dropdown** menu that will be visible with smaller display sizes.

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import “**menu**”:

```
...
@import "menu";
```

The main menu will look like this when we add more items to it:



8. JavaScript And Dropdown Menu

Next we add a **Dropdown** menu for *smaller screen sizes*.

Dropdown Markup

Edit "**base.html**" in *quarkblob/project/base/templates/base/* and add markup for *hamburger* menu toggle and **dropdown** menu below the **main menu**:

```
<ul class="menu">...</ul>
    <img onclick="myToggle()" class="mobile-menu toggle show-
mobile" alt="men\
u" src="{% static 'base/images/hamburger.png' %}">

        <ul id="dropdown" class="dropdown">
            <li class="dropdown-li
                   dropdown-li--active">
                <a class="dropdown-li-a" href="/">Home</a>
            </li>
            <li class="dropdown-li">
                <a class="dropdown-li-a"
href="/about/">About</a>
            </li>
        </ul>
```

"**onclick="myToggle()""** will run "**myToggle**" function when we click the toggle.

Dropdown Styling

Create "**dropdown.scss**" in *quarkblob/project/base/static/base/css/*:

```
$dropdown-menu-color: #aaa;
$dropdown-menu-active-color: #fff;
$dropdown-menu-bg-color: #333;
```

```
.toggle {
  position: absolute;
  top: 1em;
  right: 1em;
  border: none;
  display: inherit;
  width: 20px;
  @include for-desktop-up {
    display: none;
  }
}

.dropdown {
  display: none;
  float: right;
  width: 200px;
  font-size: 0.8em;
  background-color: $dropdown-menu-bg-color;
  padding: 1em 2em;
  list-style: none;
  position: absolute;
  top: 38px;
  right: 0;
  z-index: 200;
  text-align: right;
  border-radius: 2px;
  border: 1px solid #444;
  @include for-tablet-landscape-up {
    overflow-y: inherit;
    height: auto;
  }
  @include for-big-desktop-up {
    display: none;
  }
  &-li {
    &-a {
      color: $dropdown-menu-color;
      text-decoration: none;
      padding: 0.3em 0;
      display: inline-block;
      &:hover {
        color: $dropdown-menu-active-color;
      }
    }
    &--active {
      a {
```

```
        color: $dropdown-menu-active-color;
        font-size: 1.1em;
      }
    }
    &--last {
      border: none;
    }
  }
  &--show {
    display: block;
  }
}
```

In SASS you can define **variables** like this:

```
$dropdown-menu-color: #aaa;
```

Dropdown JavaScript

Create "**dropdown.js**" in *quarkblob/project/base/static/base/**js/***:

```
function myToggle()
{

document.getElementById("dropdown").classList.toggle("dropdown-
-show");
}

window.onclick = function(event) {
    if (!event.target.matches('.toggle')) {
        var dropdowns =
document.getElementsByClassName("dropdown");
        for (var i = 0; i < dropdowns.length; i++) {
            var openDropdown = dropdowns[i];
            if (openDropdown.classList.contains('dropdown--
show')) {
                openDropdown.classList.remove('dropdown--
show');
            }
        }
    }
};
```

“**myToggle()**” toggles the “**dropdown–show**” class when you click the *hamburger* icon. This *hides* or *reveals* the dropdown menu.

The code below it will **hide** the menu if you click **outside** it.

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/*:

Include **dropdown**:

```
...
@import "menu";
@import "dropdown";
```

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and load the **dropdown.js** script:

```
<head>
...
    <link rel="stylesheet" href="{% static 'base/css/style.css'
%}">
    <script src="{% static 'base/js/dropdown.js' %}"></script>
</head>
```

Now you can **toggle** the menu in smaller display sizes:

> “**JAVASCRIPT**” is a big topic on its own that I won’t be covering in any detail in this book. But now you should be able to load and write your own scripts to extend the website frontend functionality.

9. Blog Posts

Let's add an *app* to provide the **blogging** functionality.

Post Model

Create a **blog** app:

```
python manage.py startapp blog
```

Edit "**settings.py**" in *quarkblob/project/project/*, add the blog app to "**INSTALLED_APPS**" list and setup "**MEDIA_URL**" and "**MEDIA_ROOT**":

```
INSTALLED_APPS = [
...
    'django.contrib.staticfiles',
    'base',
    'blog',
]
MEDIA_URL = '/media/'
MEDIA_ROOT = 'media/'
```

> "**MEDIA_URL**" is the **url** from where we serve the media files. Post images will be served from paths like "**/media/post_images/01.png**".

> "**MEDIA_ROOT**" is the **physical** location for the media files. In development we use the folder "*/quarkblob/media/*". In production we will use another location (check the *Deployment* chapter).

Create the media folder in “**quarkblob/media/**”.

Edit “**models.py**” in *quarkblob/project/blog/* and *replace* the contents with these lines:

```
from django.db import models
from django.urls import reverse
from django.utils.text import slugify

class Post(models.Model):
    title = models.CharField(max_length=255, default='',
unique=True)
    slug = models.SlugField(default='', blank=True)
    post_date = models.DateTimeField(auto_now_add=True,
null=True)
    updated_date = models.DateTimeField(auto_now=True,
null=True)
    description = models.TextField(default='', blank=True)
    body = models.TextField(default='', blank=True)
    image = models.ImageField(default='', blank=True,
upload_to='post_images')

    class Meta:
        ordering = ['-post_date']

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super().save(*args, **kwargs)

    def __str__(self):
        return '%s' % self.title

    def get_absolute_url(self):
        return reverse('item', args=[str(self.slug)])
```

“**title**” is a *CharField* that we use to store **smaller strings**.

- “**max_length**” is the only *required* argument.
- We use an *empty* string as “**default**” value.

- We don’t want to have *duplicate* titles so we define “**unique=True**”.

“**slug**” is a short label that we use in **urls**.

- In “**save**” method we create the *slug* from the title using **slugify**: https://docs.djangoproject.com/en/dev/ref/utils/#django.utils.text.s

So now when we save an item with the title “**Hello World**”, the slug will be “**hello-world**”.

“**blank=True**” means that the field can be left **empty** in the *form*.

“**null=True**” will store empty values as **NULL** in database. Avoid using this with *string-based* fields like *CharField* and *TextField*.

“**post_date**” and **updated_date**” uses a *DateTimeField* to store **date** and **time** information.

- “**auto_now_add**” sets the date as *now* when you **first** create the object.
- “**auto_now**” to sets the date info **any time** the object is saved.

“**description**” and “**body**” uses a *TextField* to save larger texts. The default representation for this in forms is **textarea**.

“**image**” uses an *ImageField* to allow **image uploads**.

- “**upload_to=’post_images’**” defines the location for uploaded images. This will be inside the **media** folder.

ImageField requires the **pillow** package so let’s install it:

```
pip install pillow
```

> “**PILLOW**” package adds **image processing cababilities**.

“**Meta**” class contains data that is **not** *field data*. This includes things like **ordering** options or extra **permissions**.

- “**ordering = [‘-post_date’]**” means that we are ordering the *latest* items first. “**-**” indicates *descending* order.

“**save**” is a *built-in* method that *saves* the object to the *database*. In here we **override** it to create a *slug*.

- It’s important to call the **superclass** method with `super().save(*args, **kwargs)`. This calls the original save() method so the object will be saved to the database.
- It’s also important to pass in the **arguments** with “*args” and “**kwargs”.

“**__str__**” method is used to return a **human-readable** representation of the object. This object *method* is called whenever you call “**str()**” on an object.

- This is why we can use “**{{ object }}**” instead of “*{{ object.title }}*” to print out the title in *templates*.
- This is also used in the *admin* area to show the object titles.

“**get_absolute_url()**” defines how to get an **url** to the object.

- We use “**reverse()**” method to get the path based on the **url**. This url will be defined *later* with “*path(‘<slug:slug>/’, views.post, name=’**item**’)*”.
- “**args=[str(self.slug)]**” sends the **slug** as an *argument* to the *url*.

Makemigrations & Migrate

Run **migrations**:

```
python manage.py makemigrations && python manage.py migrate
```

"**makemigrations**" creates new **migration** files based on your models.

"**migrate**" *applies* or *unapplies* the changes based on the migration files. The first time you run it, you can see bunch of database tables created, including a table for the **Blog** items.

> "**MIGRATIONS**" | You can think **migrations** as a **version control system** for your database *schema*.

Models & Database

"**Model**" contains fields and behaviours for the **data** we are storing. Models are usually mapped to a *database* **table**. Each *attribute* like *title* or *slug* represents a *database* **field**.

Through *models* Django gives us an "**API**" to interact with the database. You can *test* this by opening the *Python interactive shell*:

```
python manage.py shell
>>> from blog.models import Post
>>> p = Post(title="My post")
>>> p.save()
>>> p.title
'My post'
```

This will actually save a new *post* item to the database.

“**Ctrl + D**” *exits* the shell.

Admin

But let’s use the **admin area** for creating more content.

> “**ADMIN AREA**” | Django provides a default **administration** area in **/admin/** that covers basic management needs. This includes **adding and editing** content, managing users etc. It is usually used by the site *administrators* and *editors*. You can also create your **own custom admin app** or modify the existing one. Later we will expose **custom blog forms** to *end-users*.

Create a **superuser**:

```
python manage.py createsuperuser
```

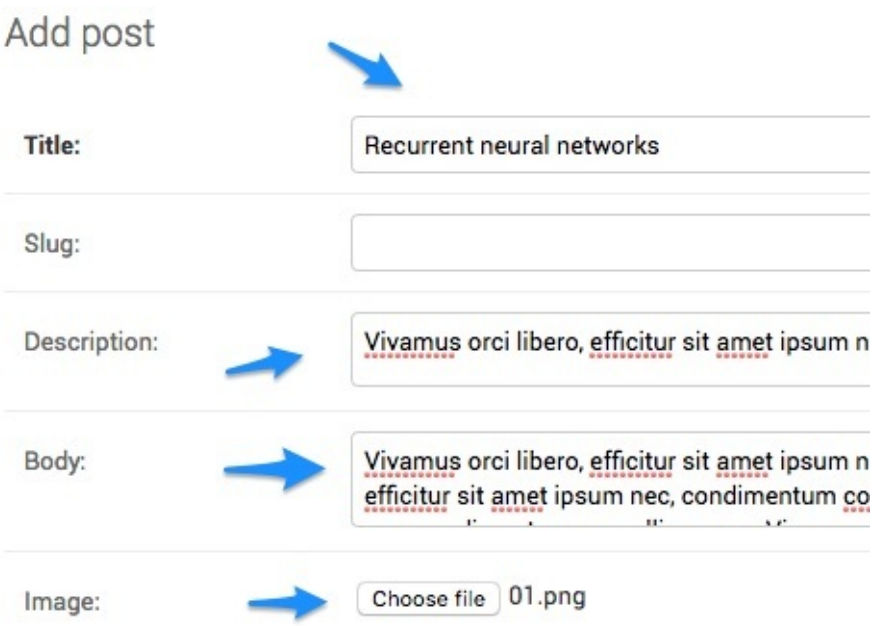
> “**SUPERUSER**” has **all permissions**. You can think this as your main **administrator** account.

Edit “**admin.py**” in *quarkblob/project/blog/* and **register** *Post*:

```
from django.contrib import admin

from blog.models import Post

admin.site.register(Post)
```

This allows us to add and edit **post** objects from the **admin** app.

Visit **/admin/**, login and add a **post**:



You can leave the slug empty. It will be generated automatically from the title.

Post View

Edit "**views.py**" in *quarkblob/project/blog/* and add these lines:

```
from django.shortcuts import render, get_object_or_404
from blog.models import Post

def post(request, slug=None):

    item = get_object_or_404(Post, slug=slug)

    return render(request, 'blog/post.html', {'item': item,
                                              'title': item,
                                              })
```

We use "**get_object_or_404**" to find a *Post* object that has a specific slug. If it doesn't find it, we will get "**404**" (Not Found) error. If an object is found, we will pass it to the template as **item**, along with the **title**.

> We could use the "**item**" variable in the template for the *title*, but you might want to build a different title in here. That's why I separated them.

Post Urls

Edit "**urls.py**" in *quarkblob/project/project/*, include "**blog.urls**" and setup "**static**" function so we can **serve media files** while running the *development server*:

```
from django.conf.urls.static import static
from django.urls import path, include
from project import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),
    path('', include('base.urls'))
] + static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

"**blog/**" will now be a **base** path for all blog app related urls.

> "**STATIC**" function will only work when the "**DEBUG**" variable is set to **True** in *settings.py* file. It doesn't affect media file serving in the production site. Read more: https://docs.djangoproject.com/en/dev/howto/static-files/#serving-files-uploaded-by-a-user-during-development. See *Deployment* chapter on how to serve media files in the **live** environment.

Create "**urls.py**" in *quarkblob/project/blog/*:

```
from django.urls import path
from blog import views
```

```
urlpatterns = [
    path('<slug:slug>/', views.post, name='item'),
]
```

So now blog posts will be found in urls like **/blog/recurrent-neural-networks/**.

> "**CAPTURING VALUES**" | We can use **angle brackets** to capture values. "**<slug:slug>**" means that we are capturing a *slug* parameter. The first *slug* is an optional **converter type**. We could just define "**<slug>**" but that would match *any string*. Now we are expecting a *slug*. "**<int:id>**" would only *match* an *integer*. Read more: https://docs.djangoproject.com/en/dev/topics/http/urls/#path-converters

Post Template

Create "**post.html**" in *quarkblob/project/blog/**templates/blog/***:

```
{% extends 'base/base.html' %}

{% block content %}

    <div class="post">
        <span class="post-author">By <strong>admin</strong>
</span>
        <span class="post-date">{{ item.post_date |date:"M d,
Y"}}</span>
        <div class="post-description clearfix">{{
item.description }}</div>
        {% if item.image %}
            <img class="post-image" src="{{ item.image.url }}">
        {% endif %}
        <div class="post-body">{{ item.body }}</div>
    </div>

{% endblock %}
```

> “**USERNAME**” | We *hard-code* the username as “**admin**” for now. Later we will add an “**author**” field to the post model.

> “**FILTERS**” | You can modify output using *built-in* or *custom* **filters**. Use **pipe** “|” to apply a filter. “**{{ item.post_date|date:”M d, Y”}}**” formats the date to the given format. Read more: https://docs.djangoproject.com/en/dev/ref/templates/builtins/#date.

Restart the development server.

Go to the *admin* area and **edit** a post in **admin/blog/post/1/change/**.

Click “**VIEW ON SITE**” on the top right. Django adds “*magically*” this link if you define “**get_absolute_url”** method in the object model. This will take you to a path like “**blog/recurrent-neural-networks/**”.

H1 & Title Tags

Let’s setup titles a bit differently.

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and change the **title** tag and add **h1** block:

```
~~<title>QuarkBlob</title>~~
<title>{% block title %}Home{% endblock %} | QuarkBlob</title>

...
<div class="main">
    {% block h1 %}{% endblock %}
    {% block content %}{% endblock %}
</div>
```

Now we can **override** the *title* and *h1* block from child templates. If we don't setup **h1** block from the child template, then we don't have the "**<h1>**" tag on the page at all. This is useful for pages like *homepage* where we don't want to show any main heading.

> "**TITLE TAG**" | Content of the "**<title>**" tag will show up in the browser **tab** name.

Edit "**post.html**" in *quarkblob/project/blog/templates/blog/*:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}
```

> "**TITLE**" | In general we are preparing the "**{{ title }}**" variable in the **view**. That way we can possibly write some custom logic to generate the title if needed. This is just one way to do it. You could as well *hard-code* these blocks here.

Post Styling

Create "**post.scss**" in *quarkblob/project/base/static/base/css/*:

```
.post {
  padding-bottom: 1em;
  &-date {
    margin: 0.5em 0;
  }
  &-description {
    font-size: 1.3em;
    margin: 0.7em 0;
  }
```

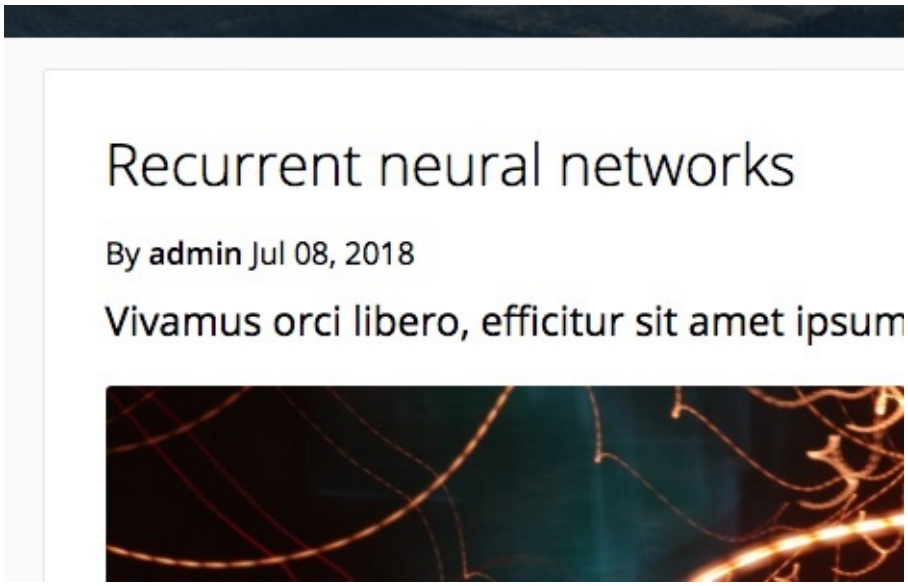
```
  &-body {
    margin: 0.3em 0;
    line-height: 1.5em;
  }
  &-image {
    margin: 0.5em 0;
    width: 100%;
    @include for-999-up {
      width: auto;
    }
  }
}
```

We want the blog "**image**" to fill full width when using smaller display sizes. But in bigger sizes we show the *default* width using "**width: auto**".

Open **style.scss** in *quarkblob/project/base/static/base/css/* and import "**post**":

```
...
@import "post";
```

Visit a blog post page and you should see something like this:



10. Responsive And Reusable Grid

In this chapter we create a **responsive** grid that will be used in *home*, *search* and *tag* pages.

Grid Template

Create "**_grid.html**" in *quarkblob/project/base/templates/base/*:

```
{% load static %}
<div class="grid items">
    {% for item in items %}
        <a class="col items-item" href="{{
item.get_absolute_url }}">
            <div class="items-item-content">
                <div class="items-item-title">
                    <div class="items-item-title-content">
                        {{ item.title | truncatechars:45 }}
                    </div>
                </div>
                {% if item.image %}
                    <img class="items-item-image" src="{{
item.image.url }}">
                {% else %}
                    <img class="items-item-image" src="{%
static 'base/images\
/no-image.png' %}">
                {% endif %}
                <div class="items-item-description">
                    {{ item.description | truncatechars:70 }}
                </div>
            </div>
        </a>
    {% endfor %}
    <div class="clearfix"></div>
</div>
```

> “**INCLUDED TEMPLATES**” | I use underscore prefix “**_**” for templates to be **included** in other templates.

> “**DRY**” | The **grid** template is now available to be **re-used** throughout the site. You don’t want to be *repeating* yourself too much. This is called the **DRY** principle: **D**on’t **R**epeat **Y**ourself. We only have to *feed* it the **items** we want to show in the grid.

Edit “**home.html**” in *quarkblob/project/base/templates/base/* and replace the contents with this:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block content %}

    {% include 'base/_grid.html' %}

{% endblock %}
```

Notice that we don’t specify the **h1** block because we don’t want to show that on the *homepage*.

Edit “**views.py**” in *quarkblob/project/base/*, fetch **all** post objects and pass them to the template:

```
from django.shortcuts import render

from blog.models import Post

def home(request):
    items = Post.objects.all()
    return render(request, 'base/home.html')
    return render(request, 'base/home.html', {'items': items,
```

```
'title': 'Home'\
})
```

> “**DATABASE ABSTRACTION**” | Once we have created the **Post** *model*, we have an **API** to interact with the database. “**Post.objects.all()**” gives us access to **all** *Post* items.

Grid Styling

Add “**grid.scss**” in *quarkblob/project/base/static/base/css/*:

```
.col {
  float: left;
}

@include for-phone-only {
  .col {
    width: 100%;
  }
}

@include for-tablet-portrait-up {
  .col {
    width: 50%;
  }
}

@include for-tablet-landscape-up {
  .col {
    width: 33.33%;
  }
}
```

> “**COLUMNS**” | “**grid.scss**” defines the *responsive columns*. In “*phone size displays*” the “**col**” class represents just **1 column**, from “*tablet portraits up*” **2 columns** and in “*largest displays*” **3**

columns. This is a bit different from systems like **Bootstrap** where different class names represent different column sizes.

"**MODERN GRID TOOLS**" | I'm creating a very *simple* grid solution here but you might want to check out **Grid**: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Basic_Concepts_of_Grid_Layout, **Flexbox**: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbo or **Susy**: http://oddbird.net/susy/.

Create "**items.scss**" in *quarkblob/project/base/static/base/css/*:

```
.items {
  &-item {
    display: block;
    &-content {
      margin: 3px;
      background-color: #fff;
      padding: 1em;
      border: 1px solid #E8E8E8;
      border-radius: 2px;
      height: 310px;
    }
    &-image {
      width: 100%;
    }
    &-title, &-description {
      padding: 0.5em 0;
      line-height: 1.5em;
    }
    &-title {
      font-size: 1.2em;
    }
  }
}
```

> “**GUTTER**” | Notice that I accomplished the *gutter* between the grid items by applying “**margin: 3px;**” for the *inner* container “**items-item-content**”.

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import the files:

```
...
@import "grid";
@import "items";
```

Hide Main Container

Let’s add an option to “**hide**” the content box wrapper:

Edit “**home.html**” in *quarkblob/project/base/templates/base/* and add **hidebox** block:

```
{% block title %}{{ title }}{% endblock %}
{% block hidebox %}main--no-box {% endblock %}
{% block content %}
```

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and add **hidebox** block:

```
<div class="main">
<div class="main {% block hidebox %}{% endblock %}">
```

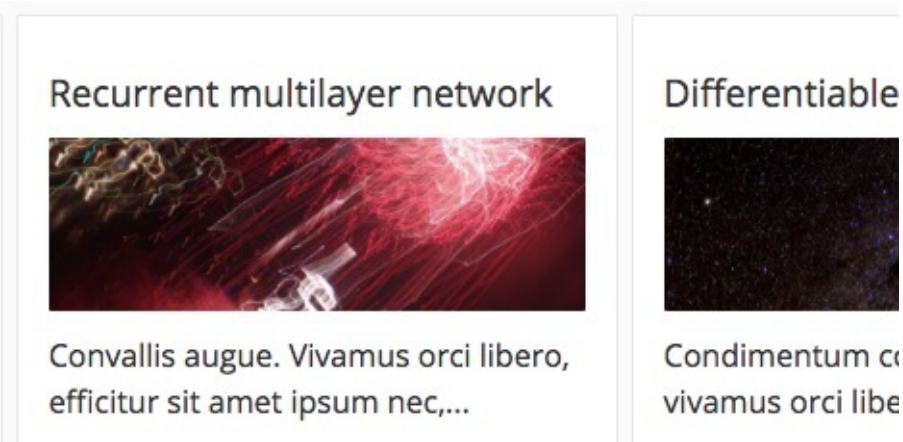
Edit “**layout.scss**” in *quarkblob/project/base/static/base/css/* and add styling for **main–no-box** class:

```
.main {

  ...

  @include for-999-up {
```

```
    width: 60em;
  }
  &--no-box {
    padding: 0;
    background-color: inherit;
    border: inherit;
  }
}
```

> “**NO-BOX**” | So now if we don’t want to show the *white* container box on some pages, we can add “**{% block hidebox %}main–no-box {% endblock %}**” to the child template. The grid items already have white background that doesn’t work well with visible background container. Arguable this adds *complexity* but it’s a **design** question in the end.

Add more items and you should see a **responsive grid**:



11. Reusable Pagination

Django provides *classes* to create basic **pagers**.

Pagination is also a good example for **reusable** templates. We can use this same template in all the pages we want to include a **pager**.

Paginator Class

Edit "**views.py**" in *quarkblob/project/base/* and change the "**home**" view:

```
from django.core.paginator import Paginator

def home(request):
    items = Post.objects.all()
    paginator = Paginator(items, 3)
    page = request.GET.get('page')
    items = paginator.get_page(page)

    return render(request, 'base/home.html', {'items': items,
                                              'title': 'Home'
                                              })
```

With "**paginator = Paginator(items, 3)**" we give **Paginator** two arguments: the *items* we want to paginate and the *number of items* we want to see on each page. I use a small number so we don't have to create so many items to test this.

"**page = request.GET.get('page')**" will get the **page** number from the *request* object. We can send it like this: "**quarkblob.com/?page=2**".

"**items = paginator.get_page(page)**" will get *only* those items that are in a particular *page*. We will not load *all* items at once. This is very

important if you have a lot of items.

"**items**" *object* will now contain helpful *attributes* and *methods* that we can use to create a custom paginator.

Simple Example

Create "**_pagination.html**" in *quarkblob/project/base/templates/base/*.

Here is a simple example that you can *start* with:

```
<div class="pagination">
    {% if items.has_previous %}
        <a href="?page=1">First</a>
        <a href="?page={{ items.previous_page_number
}}">Previous</a>
    {% endif %}

    <span>{{ items.number }}</span>
    <span">of</span>
    <span>{{ items.paginator.num_pages }}</span>

    {% if items.has_next %}
        <a href="?page={{ items.next_page_number }}">Next</a>
        <a href="?page={{ items.paginator.num_pages
}}">Last</a>
    {% endif %}
</div>
```

We can use the same "**items**" object that we use to loop through all objects. We don't have to change the "*_grid.html*" template. But we also have **additional** attributes and methods available like "**items.has_previous**" that tells us if there is a "*previous*" page available etc.

Check the official documention https://docs.djangoproject.com/en/dev/topics/pagination/ for more

info. Also check my free **Pagination tutorial** in here https://www.wdtutorials.com/django/pagination-examples/.

More Complex Example

Let's add a little bit more *detailed* example.

Edit "**_pagination.html**" in *quarkblob/project/base/templates/base/* and replace it with this:

```
{% if items.paginator.count > 1 %}

    <div class="pagination clearfix">

        {% if items.has_previous %}
            <a class="pagination-action" href="?page=1{% if
request.GET.q %}&\
q={{ request.GET.q }}{% endif %}">
                <i class="fa fa-angle-double-left" aria-
hidden="true"></i>
            </a>
            <a class="pagination-action" href="?page={{
items.previous_page_n\
umber }}{% if request.GET.q %}&q={{ request.GET.q }}{% endif
%}">
                <i class="fa fa-angle-left" aria-hidden="true">
</i>
            </a>
        {% endif %}

        {% if items.paginator.num_pages > 1 %}

            {% for num in items.paginator.page_range %}

                {% if items.number == num %}
                    <span class="pagination-number pagination-
current">{{ num\
 }}</span>
                {% elif num > items.number|add:'-3' and num <
items.number|ad\
d:'3' %}
                    <a class="pagination-number" href="?page={{
num }}{% if r\
equest.GET.q %}&q={{ request.GET.q }}{% endif %}">{{ num }}</a>
```

```
                {% endif %}

            {% endfor %}

        {% endif %}

        {% if items.has_next %}
            <a class="pagination-action" href="?page={{
items.next_page_numbe\
r }}{% if request.GET.q %}&q={{ request.GET.q }}{% endif %}">
                <i class="fa fa-angle-right" aria-
hidden="true"></i>
            </a>
            <a class="pagination-action" href="?page={{
items.paginator.num_p\
ages }}{% if request.GET.q %}&q={{ request.GET.q }}{% endif
%}">
                <i class="fa fa-angle-double-right" aria-
hidden="true"></i>
            </a>
        {% endif %}

    </div>

{% endif %}
```

Edit "**home.html**" in *quarkblob/project/base/templates/base/* and include **pagination**:

```
{% include 'base/_grid.html' %}
    {% include 'base/_pagination.html' %}
```

Pagination Styling

Let's add styling for the more *complex* example.

Create "**pagination.scss**" in *quarkblob/project/base/static/base/css/*:

```
.pagination {
  text-align: center;
  margin: 2em 0;
  &-number {
    padding: 0.3em 0.8em;
```

```
    border-radius: 2px;
    color: #fff;
    background-color: #AFAFAF;
    &:hover {
      background-color: #5EAEFD;
    }
  }
  &-current {
    background-color: #5EAEFD;
  }
  &-action {
    margin: 0 0.1em;
    display: inline-block;
    padding: 0.5em 0.5em;
    color: #B9B9B9;
    font-size: 1.3em;
    &:hover {
      color: #3354AA;
    }
  }
  &-previous, &-next {
    color: #3354AA;
  }
}
```

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import “**pagination**”:

```
...
@import "pagination";
```

Here is the end **result**:

12. Thumbnails

Loading **original** images can make the website **considerably** heavier. That's why you often want to use an image *processor* to automatically create smaller images from the original images.

Imagekit Package

> "**IMAGEKIT**" package adds *processors* for common image handling needs, like **resizing** and **cropping**.

Install **imagekit** package:

```
pip install django-imagekit
```

Edit "**settings.py**" in *quarkblob/project/project/* and add "**imagekit**" to the list:

```
INSTALLED_APPS = [
    ...
    'blog',
    'imagekit',
]
```

Edit "**models.py**" in *quarkblob/project/blog/* and add "**image_thumbnail**" and "**image_large**" fields to the "**Post**" model:

```
from imagekit.models import ImageSpecField
from pilkit.processors import ResizeToFill

class Post(models.Model):
```

```
    ...
    image = models.ImageField(default='', blank=True,
upload_to='post_images')
    image_thumbnail = ImageSpecField(source='image',
                                     processors=
[ResizeToFill(250, 100)],
                                     format='JPEG',
                                     options={'quality': 60})
    image_large = ImageSpecField(source='image',
                                 processors=[ResizeToFill(700,
250)],
                                 format='JPEG',
                                 options={'quality': 60})
```

We will use "**ResizeToFill(250, 100)**" to create thumbnails for the **grid** and "**[ResizeToFill(700, 250)**" to create bigger images for the actual **post** page.

Thumbnail Generation

Edit "**post.html**" in *quarkblob/project/blog/templates/blog/* and use "**item.image_large.url**":

```
<div class="post-description clearfix">{{ item.description }}
</div>
{% if item.image %}
    <img class="post-image" src="{{ item.image.url }}">
    <img class="post-image" src="{{ item.image_large.url }}">
{% endif %}
<div class="post-body">{{ item.body }}</div>
```

Edit "**_grid.html**" in *quarkblob/project/base/templates/base/* and use "**item.image_thumbnail.url**":

```
{% if item.image %}
    <img class="items-item-image" src="{{ item.image.url }}">
    <img class="items-item-image" src="{{
item.image_thumbmail.url }}">
{% else %}
    <img class="items-item-image" src="{% static
'base/images/no-image.png' %\
```

```
}">
{% endif %}
```

Now when you visit the “**home**” or “**post**” page, the thumbnails are generated and served from locations like this:

“**media/CACHE/images/post_images/02..e8b.jpg**”.

Everything will look pretty much the same but the loaded images are possibly **much** smaller.

13. Tags And Relationships

Django provides three *main* relationship types:

"**ManyToManyField**" field defines a **many-to-many** relationship. We will use this with **tags** field. Post can have **many** tags and tag can relate to **many** posts.

"**ForeignKey**" defines a **many-to-one** relationship. We will use this with **author** field. Post can have only **one** author but author can relate to **many** posts.

"**OneToOneField**" defines a **one-to-one** relationship. We will use this with user **profile**. Profile can relate to only **one** user and user can relate to only **one** profile.

Tag Model & ManyToManyField

Edit "**models.py**" in *quarkblob/project/blog/* and add "**Tag**" class and "**tags**" field:

```
class Tag(models.Model):
    title = models.CharField(max_length=255, default='')
    slug = models.SlugField(default='', blank=True)

    class Meta:
        ordering = ['title']

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super().save(*args, **kwargs)

    def __str__(self):
        return '%s' % self.title
```

```
    def get_absolute_url(self):
        return reverse('tag', args=[str(self.slug)])

class Post(models.Model):
    ...
    tags = models.ManyToManyField(Tag, blank=True)
```

"**title**", "**slug**" and rest of the *attributes* are defined exactly the same as in the *Post* model.

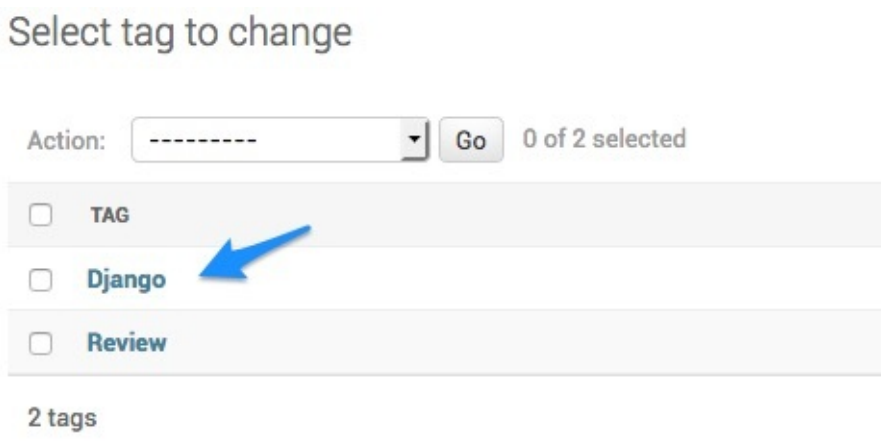
Run **migrations**:

```
python manage.py makemigrations && python manage.py migrate
```

Edit "**admin.py**" in *quarkblob/project/blog/* and register "**Tag**":

```
from django.contrib import admin
from blog.models import Post, Tag

admin.site.register(Post)
admin.site.register(Tag)
```

Visit **admin/blog/tag/** and add some **tags**:



Edit a post and select some tags for it. You can also add tags by hitting the green "+" next to the tags field.

Post Page Tags

Edit "**post.html**" in *quarkblob/project/blog/templates/blog/* and loop through the **tags**:

```
<div class="post">
    ...
    <div class="post-body">{{ item.body }}</div>
    {% if item.tags %}
        <ul class="tags">
            {% for tag in item.tags.all %}
                <li class="tags-tag">
                    <a class="tags-tag-a" href="{% url 'tag'
tag.slug %}">{{ \
tag }}</a>
                </li>
            {% endfor %}
        </ul>
    {% endif %}
</div>
```

Here we are sending the tag **slug** as parameter so the url tag can display the right url: "**{% url 'tag' tag.slug %}**".

Tag View

Edit "**views.py**" in *quarkblob/project/base/* and add the **tag** method:

```
from django.core.paginator import Paginator
from django.shortcuts import render, get_object_or_404
from blog.models import Post, Tag

def tag(request, slug=None):
    _tag = get_object_or_404(Tag, slug=slug)
    items = Post.objects.filter(tags__slug=slug)
    title = 'Items tagged with "%s"' % _tag
    return render(request, 'base/tag.html', {'items': items,
                                             'tag': _tag,
                                             'title': title
                                             })
```

> “**RELATIONSHIP LOOKUPS**” | In “**Post.objects.filter(tags__slug=slug)**” we use **double underscores** to get all **post** objects that has a **tag** with a specific **slug**.

We use “**title = ‘Items tagged with “%s”’ % _tag**” to build the *title* using the given tag.

Tag Grid

Create “**tag.html**” in */quarkblob/project/base/templates/base/* and add these lines:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block hidebox %}main--no-box{% endblock %}
{% block content %}

    <div class="tag page">
        {% include 'base/_grid.html' %}
    </div>

{% endblock %}
```

Edit “**urls.py**” in *quarkblob/project/project/* and add the **tags** url:

```
from django.contrib import admin
from django.urls import path, include

from base import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('tags/<slug:slug>/', views.tag, name='tag'),
    path('', include('base.urls')),
]
```

Tag Styling

Create “**tags.scss**” in *quarkblob/project/base/static/base/css/* and add these lines:

```
.tags {
  list-style: none;
  padding-left: 0;
  &-tag {
    display: inline-block;
    padding: 0.6em 0.8em;
    background-color: #3C98F5;
    color: #fff;
    margin-right: 0.5em;
    border-radius: 2px;
    &-a {
      color: #fff;
    }
    &:hover {
      background-color: darken(#3C98F5, 10%);
    }
  }
}
```

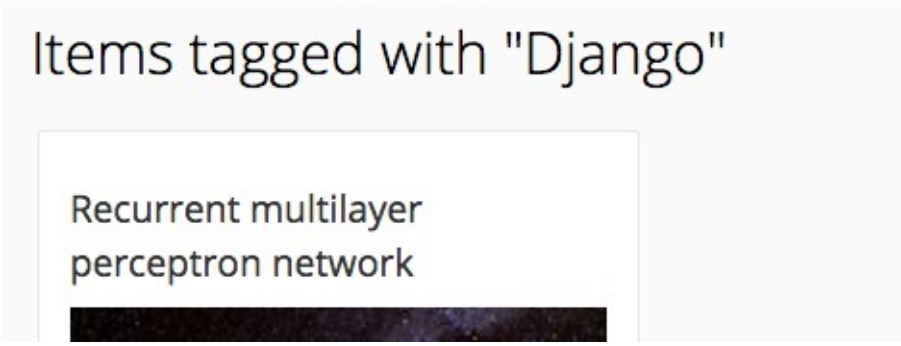
> “**DARKEN**” is a useful *function* in **SASS** to make color darker.

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import **tags**:

```
...
@import "tags";
```

Visit a **post** with tags and you should see the list:

If you click the tag, you will see **all posts** *tagged* with that tag:



"**REUSING GRID**" | Notice that we are reusing the **grid** template. You could also add **pagination** to this page using the code from the **post** view and template.

14. Custom Post Form

ModelForm

For now we have used the **/admin/** area to **create** and **edit** posts. This can be suitable for *editors* and *administrators* but you might also want to *expose* specific forms to other users.

Dealing with forms can be a complicated process. Fortunately Django offers some tools to automate most of it. In the center of it is the **Form** class.

> "**FORM**" *class* describes a form and determines how it functions and appears.

In this chapter we will be focusing in a helper class called **ModelForm**.

> "**MODELFORM**" allows us to create forms from Django models. We have already defined the necessary **fields** in **Post** model so we don't want to do that again. This is where *ModelForm* comes handy.

Post Form Template

Create "**post_form.html**" in */quarkblob/project/blog/templates/blog/*:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}

<form class="post-form" method="POST" enctype="multipart/form-
data" action="">
    {% csrf_token %}
    {{ form }}
    <input class="post-form-submit submit" type="submit"
value="Save">
</form>

{% endblock %}
```

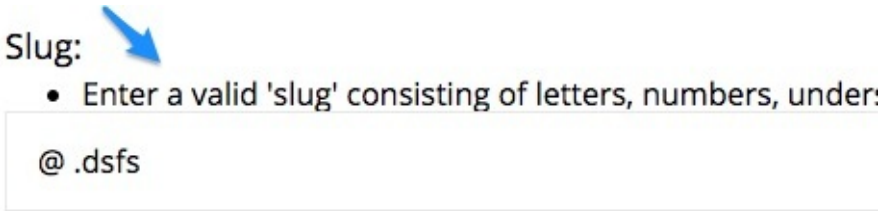
‘**enctype=”multipart/form-data”**’ allows us to send **files** with the form.

Use “**{% csrf_token %}**” for “**POST**” forms that target *internal* paths. Don’t use it for forms that target *external* paths. This adds protection against **Cross Site Request Forgeries**.

‘**action=”“**’ means that the form is sending data to the current URL.

“**{{ form }}**” is an easy way to print out **all** fields, labels and attributes. It will also print **error** messages.

Here is an example of *validation error*:



Post Form Urls

Edit “**urls.py**” in *quarkblob/project/blog/* and add “**urls**” for *adding* and *editing* post items:

```
urlpatterns = [
    path('<slug:slug>/', views.post, name='item'),
    path('add/post/', views.add_post, name='add_post'),
    path('edit/post/<int:pk>/', views.edit_post,
name='edit_post'),
]
```

Forms.py

Create "**forms.py**" in *quarkblob/project/blog/*:

```
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = '__all__'
```

"**class PostForm(ModelForm)**" defines a class that inherits from *ModelForm*. In the "**Meta**" class we define the *model* and *fields* we want to show in the form. "**'__all__'**" means… all fields.

Post Form Views

Edit "**views.py**" in *quarkblob/project/blog/* and add "**add_post**" view:

```
from django.shortcuts import render, get_object_or_404,
redirect
from blog.forms import PostForm

from blog.models import Post

def add_post(request):
    if request.method == "POST":
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            item = form.save(commit=False)
            item.save()
            form.save_m2m()
            return redirect(item.get_absolute_url())
    else:
```

```
        form = PostForm()

    return render(request, 'blog/post_form.html', {'form':
form,
                                                    'title':
'Add post',
                                                    })
```

In “**add_post**” function we first check if the “**method**” found in the *HttpRequest* object is **POST**. If the method **is not** POST, then we create a form *object* and pass it to the template. This is the default action when we first visit the page and see an **empty** form.

If the method **is** POST, then we create a “**form**” object using the **data** provided by the from.

“**request.POST**” contains post **data** like *title* for the blog post.

“**request.FILES**” contains **uploaded files**. In our case that will contain the post *image*.

If the **form** instance is tied to a set of data, we call it a **bound** form. This simply means that we have posted data with the HTML form and that data is now ready to be validated.

We use “**form.is_valid()**” method to validate all fields. Django has some *built-in* validators that it will use *internally*. For example “**validate_slug**” makes sure that a value consists of only letters, numbers, underscores or hyphens.

ModelForm object has a “**save**” method that *creates* and *saves* objects bound to that form. “**commit=False**” means that the *post_item* object is **created** but not saved. This way we can do *custom processing* before we save the item to the database. We will use this later to add the *current* logged-in user as the **author** of the post. That’s why we have to run “**item.save()**” also to actually save the item:

```
item = form.save(commit=False)
// we will store author here..
item.save()
```

We use "**form.save_m2m()**" to save the many-to-many form data (tags data). When you use "**commit=False**" in "**form.save(commit=False)**" these relationships are *not* saved. But Django adds a "**save_m2m()**" method that we can use to save the relationship data: https://docs.djangoproject.com/en/dev/topics/forms/modelforms/#the-save-method.

"**return redirect(item.get_absolute_url())**" will take us to the **post page** after the item has been saved. It's easy to get the path to the item because we defined the "**get_absolute_url()**" method in the **Post model**.

Edit "**views.py**" in *quarkblob/project/blog/* and add "**edit_post**" view:

```
def edit_post(request, pk=None):
    item = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, request.FILES,
instance=item)
        if form.is_valid():
            form.save()
            return redirect(item.get_absolute_url())
    else:
        form = PostForm(instance=item)

    title = 'Edit: %s' % item
    return render(request, 'blog/post_form.html', {'form':
form,
                                                   'item':
item,
                                                   'title':
title,
                                                   })
```

In “**edit_post**” we use “**item = get_object_or_404(Post, pk=pk)**” to check if a post with specific **primary key** (pk) exists. If so, we create the **form** object and relate it to the corresponding post object.

“**instance=item**” will **attach** a model instance to the form so when we do “**form.save()**”, it saves the object. This will also show us the current object data with the form so we can **edit** it.

We don’t need “**form.save_m2m()**” because we don’t use “**commit=False**” with “**form.save()**”.

Here is more *concise* version of the form above that you can use:

```
def edit_post(request, pk=None):
    item = get_object_or_404(Post, pk=pk)
    form = PostForm(request.POST or None, request.FILES or
None, instance=ite\
m)
    if form.is_valid():
        form.save()
        return redirect(item.get_absolute_url())
    title = 'Edit: %s' % item
    return render(request, 'blog/post_form.html', {'form':
form,
                                                   'item':
item,
                                                   'title':
title,
                                                   })
```

In here we have one *less* “**if/else**” structure. Use “**request.POST or None**” instead of **request.POST** or the *edit* form will be empty when editing a post. This is because **request.POST** is **empty** when we first arrive to the page. We need to send **None** argument instead.

Form Styling

I like to style **all** forms in one place.

Create “**forms.scss**” in *quarkblob/project/base/static/base/css/*:

```
form {
  label {
    font-size: 18px;
    display: block;
    padding-bottom: 0.3em;
    margin-top: 1em;
  }

  .helptext {
    display: inline-block;
    margin-top: 0.2em;
    font-style: italic;
  }

  ul {
    margin: 0;
  }

  p {
    margin: 0;
  }

  input, textarea, select {
    width: 100%;
    padding: 1em;
    border: 1px solid #E8E8E8;
  }

  textarea#id_description {
    height: 100px;
  }
}

.submit {
  display: inline-block;
  width: auto;
  background-color: #EF73C4;
  padding: 0.8em 1em;
  margin: 1em 0;
  border-radius: 3px;
  color: #fff;
  font-size: 14px;
  letter-spacing: 1px;
  text-transform: uppercase;
```

```
  border: none;
  &:hover {
    background-color: darken(#EF73C4, 10%)
  }
}
```

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import “**forms**”:

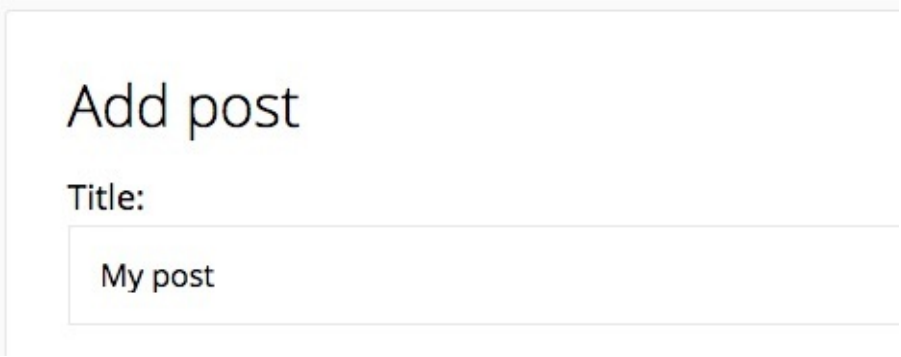
```
...
@import "forms";
```

Add Post Link

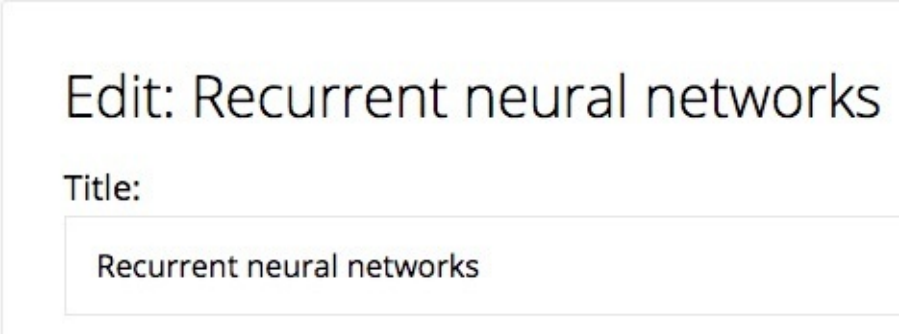
Edit “**base.html**” in *quarkblob/project/base/templates/base/* and add “**Add Post**” link between **Home** and **About** links:

```
<li class="menu-li">
    <a class="menu-li-a {% url 'add_post' as add_post_url %} {%
if request.ge\
t_full_path == add_post_url %} menu-li-a--active{% endif %}"
href="{% url 'ad\
d_post' %}">Add post</a>
</li>
```

Now you can allow visitors to **Add** posts in “**/blog/add/post/**”:



And **Edit** posts in “**/blog/edit/post/<pk>/**”:



In the *Authentication* chapter we will **restrict** access to these *views*.

15. Custom Search

In this chapter we create a basic **search** feature.

Search App

Create the **search** app:

```
python manage.py startapp search
```

Edit "**settings.py**" in *quarkblob/project/project/* and add **search** to the "**INSTALLED_APPS**" list:

```
INSTALLED_APPS = [
    'imagekit',
    'search',
]
```

Search Templates

Create "**_search_form.html**" in *quarkblob/project/search/templates/search/*:

```
<form class="search" method="GET" action="/search/">
    <i class="fa fa-search search-icon"></i>
    <input id="q" name="q" value="{{ request.GET.q }}"
class="search-input" t\
ype="text" placeholder="Search...">
</form>
```

We use **underscore** to indicate that this is an **include** to be used in multiple places. We want to add the search form on the *home* **and** *search* page.

“**{{ request.GET.q }}**” gets the current search **word** so it stays in the input field after you submit the form.

Create “**search.html**” in *quarkblob/project/search/templates/search/*:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block hidebox %}main--no-box{% endblock %}
{% block content %}
    {% include 'search/_search_form.html' %}
    {% if items %}
        <p class="search-count">Found <strong>{{
items.paginator.count }}</st\
rong> result{{ items.paginator.count|pluralize}}.</p>
        {% include 'base/_grid.html' %}
    {% elif request.GET.q %}
        <p class="search-no-results">No results for "<strong>{{
request.GET.q\
 }}</strong>"</p>
    {% endif %}
    {% include 'base/_pagination.html' %}
{% endblock %}
```

In “**{{ items.paginator.count|pluralize}}**” *pluralize* returns a plural suffix if the *count* value is not 1. By default this is ‘**s**’.

Edit “**home.html**” in *quarkblob/project/base/templates/base/* and include “**_search_form.html**”:

```
    {% include 'search/_search_form.html' %}
    {% include 'base/_grid.html' %}
```

Search View

Create “**views.py**” in *quarkblob/project/search/*:

```
from django.core.paginator import Paginator
from django.shortcuts import render

from blog.models import Post
```

```
def search(request):

    q = request.GET.get('q', None)
    items = ''

    if q is None or q is "":
        items = Post.objects.all()
    elif q is not None:
        items = Post.objects.filter(title__contains=q)

    paginator = Paginator(items, 3)
    page = request.GET.get('page')
    items = paginator.get_page(page)

    title = "Search"

    return render(request, 'search/search.html', {'items':
items,
                                                  'title':
title})
```

"**q = request.GET.get('q', None)**" gets the search word.

If there is no search word, we get all items. Else we search the object titles with "**Post.objects.filter(title__contains=q)**".

Search Urls

Create "**urls.py**" in *quarkblob/project/search/*:

```
from django.urls import path

from search import views

urlpatterns = [
    path('', views.search, name='search'),
]
```

Edit "**urls.py**" in *quarkblob/project/project/* and include "**search.urls**":

```
urlpatterns = [
    ...
    path('tags/<slug:slug>/', views.tag, name='tag'),
    path('search/', include('search.urls')),
    path('', include('base.urls'))
]
```

Search Styling

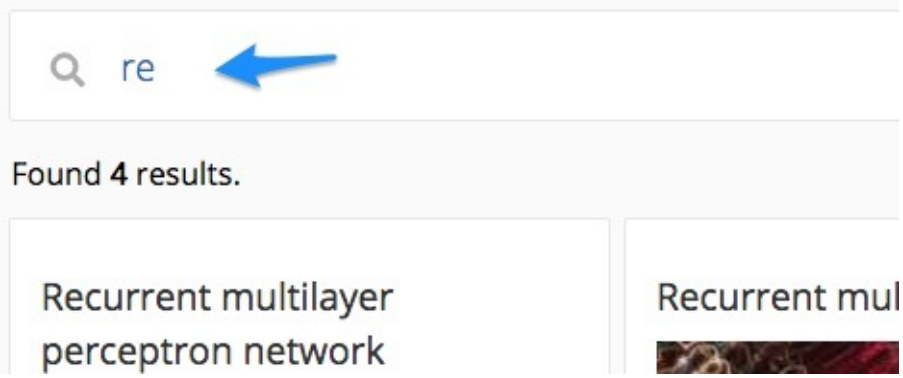
Create "**search.scss**" in *quarkblob/project/base/static/base/css/*:

```
.search {
  position: relative;
  padding: 0 0.2em;

  &-count {
    margin: 0 0 0.5em 0.3em;
  }

  &-icon {
    position: absolute;
    top: 1.2em;
    left: 1.4em;
    font-size: 18px;
    color: #ACACAC;
  }

  &-input {
    width: 100%;
    padding: 1.1em 0 1em 3em;
    margin-bottom: 0.8em;
    border-radius: 3px;
    border: 1px solid #E8E8E8;
    color: #1067BF;
    font-size: 1.2em;
    font-weight: 300;
    height: 3em;
  }
}
```

Edit "**style.scss**" in *quarkblob/project/base/static/base/css/* and import "**search**":

```
@import "search";
```

Now you can **filter** items by searching the title fields:



Check out my "**ELASTICSEARCH**" tutorial on how to start creating more advanced search feature: https://www.wdtutorials.com/django/fast-search-with-elasticsearch/.

16. Authentication

Django comes with a built-in **authentication** system. There are several built-in **forms** and **views** so we don't have to write any *extensive* backend logic. But we have to provide some **template** files.

Account Urls

Edit "**urls.py**" in *quarkblob/project/project/* and add these "**paths**":

```
path('search/', include('search.urls')),
path('accounts/register/', views.Register.as_view(),
name='register'),
path('accounts/', include('django.contrib.auth.urls')),
path('', include('base.urls'))
```

The last line will implement the following **patterns**:

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/
[name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

For **registration** we will use "**UserCreationForm**":
https://docs.djangoproject.com/en/dev/topics/auth/default/#django.con

Account Templates

Edit "**settings.py**" in *quarkblob/project/project/* and change the "**'DIRS'**" line as follows:

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        ...
    },
]
```

> "**DIRS**" defines a *list* of directories where Django should search for template files.

Create "**/templates/registration/**" folders in *quarkblob/project/* and put following "**template files**" inside:

"**login.html**" allows people to *login*:

```
{% extends 'base/base.html' %}
{% block title %}Login{% endblock %}
{% block h1 %}<h1>Login</h1>{% endblock %}

{% block content %}

    <form method="post" action="{% url 'login' %}">
        {% csrf_token %}
        {{ form.as_p }}
        <input class="submit" type="submit" value="login" />
        <input type="hidden" name="next" value="{{ next }}" />
    </form>

    <p>
        <a href="{% url 'password_reset' %}">Forgot password?
</a> |
        <a href="{% url 'register' %}">Register</a>
    </p>

{% endblock %}
```

“**{{ form.as_p }}**” will wrap the “**<label>/<input>**” pairs in “**<p>**” tags.

Form *action* “**{% url ‘login’ %}**” will point to */accounts/login/*.

“**password_reset_complete.html**” is displayed after successful *password reset*:

```
{% extends 'base/base.html' %}
{% block title %}Password reset complete{% endblock %}
{% block h1 %}<h1>Password reset complete</h1>{% endblock %}

{% block content %}
<p>Password reset successful. <a class="submit" href="{% url
'login' %}">Clic\
k this to log in.</a></p>
{% endblock %}
```

“**password_reset_confirm.html**” is a confirmation form for *changing the password*:

```
{% extends 'base/base.html' %}
{% block h1 %}<h1>Set Password</h1>{% endblock %}

{% block content %}
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <input class="submit" type="submit" value="Change Password">
</form>
{% endblock %}
```

“**password_reset_done.html**” informs user that an *email* has been sent:

```
{% extends 'base/base.html' %}
{% block h1 %}<h1>Reset Email Sent</h1>{% endblock %}

{% block content %}
    <p>Please check you inbox for instructions.</p>
{% endblock %}
```

“**password_reset_form.html**” allows user to *reset password*:

```
{% extends 'base/base.html' %}
{% block h1 %}<h1>Reset Password</h1>{% endblock %}

{% block content %}
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input class="submit" type="submit" value="Reset password">
  </form>
{% endblock %}
```

“**register.html**” allows user *registration*:

```
{% extends 'base/base.html' %}
{% block title %}Register{% endblock %}
{% block h1 %}<h1>Register</h1>{% endblock %}

{% block content %}
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button class="submit" type="submit">Register</button>
    <p>
        <a href="{% url 'password_reset' %}">Forgot password?
</a> |
        <a href="{% url 'login' %}">Login</a>
    </p>
  </form>
{% endblock %}
```

“**password_change_form.html**” allows user to *change password*:

```
{% extends 'base/base.html' %}
{% block title %}Change Password{% endblock %}
{% block header %}Change Password{% endblock %}

{% block content %}
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button class="submit" type="submit">Change
Password</button>
```

```
  </form>
{% endblock %}
```

“**password_change_done.html**” is a *confirmation* message for successful password change:

```
{% extends 'base/base.html' %}
{% block title %}Password Changed{% endblock %}
{% block header %}Password Changed{% endblock %}

{% block content %}
    <p>Your password has been changed.</p>
{% endblock %}
```

Register & Class Based Views

Thus far we have only used view **functions** like “**add_post**” or “**edit_post**”. But view can also be a “**class**”. Class based views allow more advanced scenarios like using *inheritance*.

Let’s use a *class based view* for the **sign-up** functionality.

Edit “**views.py**” in *quarkblob/project/base/* and add a “**Register**” class:

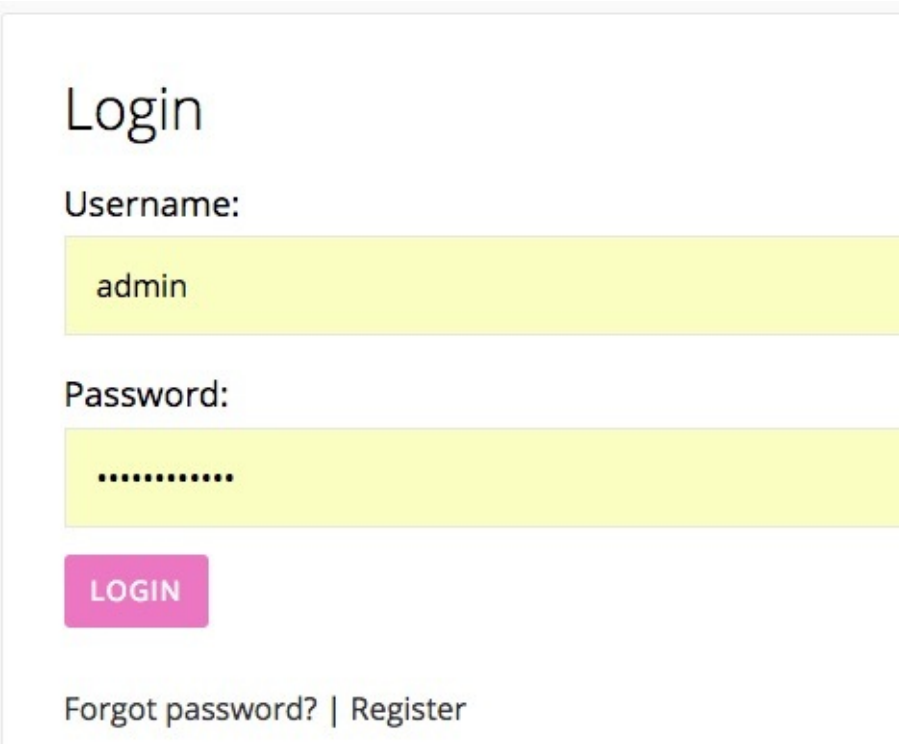
```
from django.core.paginator import Paginator
from django.shortcuts import render, get_object_or_404
from django.contrib.auth.forms import UserCreationForm
from django.urls import reverse_lazy
from django.views import generic

class Register(generic.CreateView):
    form_class = UserCreationForm
    template_name = 'registration/register.html'
    success_url = reverse_lazy('login')
```

> “**LAZY**” | Django uses *lazy* strategies in many places. **Laziness** delays computation until it’s actually required. Normally we would

use “**reverse()**” to get the path to the login page. But in this case we have to use “**reverse_lazy()**” to delay it because the urls are not loaded at this point: https://docs.djangoproject.com/en/dev/ref/urlresolvers/#django.urls.reverse

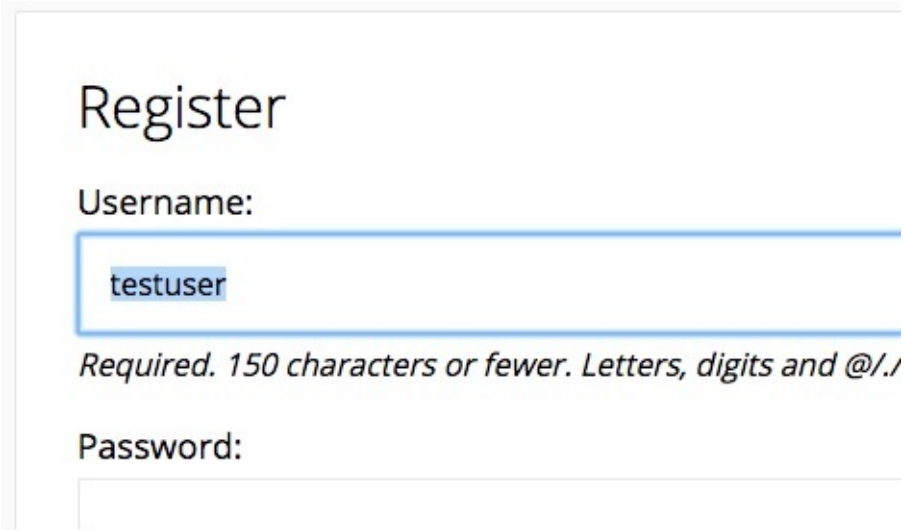
Since we have already defined general styling for forms in **forms.scss** you now have a fully functional authentication system.

“**Login**” in *accounts/login/*:

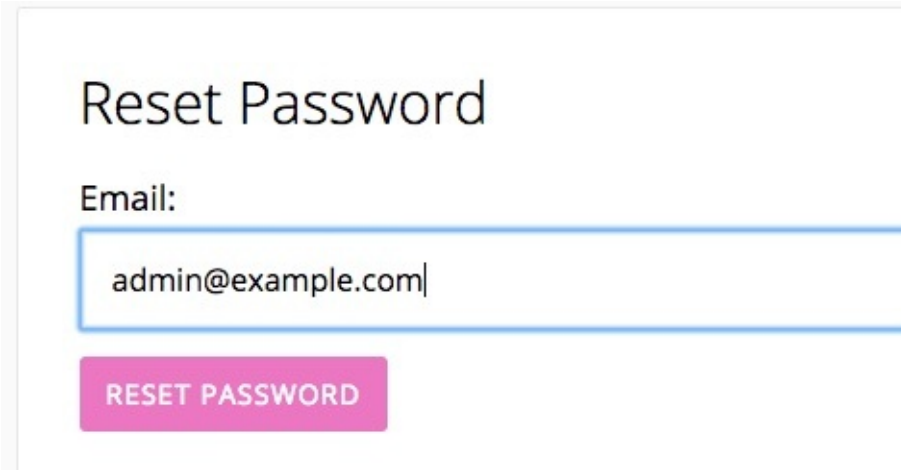


“**PROFILE**” | You will be redirected to “**accounts/profile/**” after successful login. We will implement this feature later.

“**Register**” in *accounts/register/*:



"**Reset password**" in *accounts/password_reset/*:



> "**EMAILS**" | Site is not sending any emails at this moment. We will implement this feature later.

Login & Logout Links

Edit "**base.html**" in *quarkblob/project/base/templates/base/* and add links for "**Logout**" and "**login / register**" in the *dropdown menu*:

```
<li class="dropdown-li">
    <a class="dropdown-li-a" href="/about/">About</a>
</li>
{% if request.user.is_authenticated %}
    <li class="dropdown-li">
        <a class="dropdown-li-a" href="{% url 'logout'
%}">Logout</a>
    </li>
{% else %}
      <li class="dropdown-li">
        <a class="dropdown-li-a" href="{% url 'login' %}">login
/ register</a>
    </li>
{% endif %}
```

Add also “**login / register**” link above the *slogan* markup:

```
{% if not request.user.is_authenticated %}
    <a class="header-user show-mobile" href="{% url 'login'
%}">
        login / register
    </a>
{% endif %}

<div class="header-slogan">Thoughts On <strong>Artificial
Intelligence</stron\
g>.</div>
```

We can use “**{% if not request.user.is_authenticated %}**” to check if the user is **not** logged in.

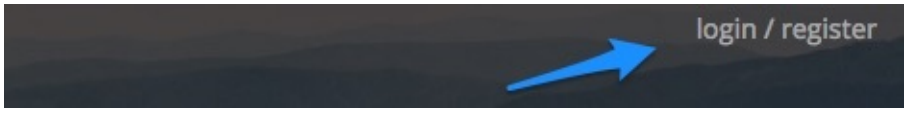
Styling

Edit “**header.scss**” in *quarkblob/project/base/static/base/css/* and add styling for the “**header-user**” element:

```
&-brand-name { ... }

  &-user {
    display: none;
    position: absolute;
    top: 0.8em;
```

```
    right: 1em;
    color: #aaa;
    font-size: 14px;
    @include for-desktop-up {
      display: inherit;
    }
    &:hover {
      color: #fff;
    }
  }

  &-slogan { ... }
```

Now you will see "**login / register**" link on the top when you are not logged in:



After we have created the **Profile** functionality, we will add an option to **edit** the user data in the "*Extend User Forms*" chapter.

> Check out my video tutorial on how to use "**ALLAUTH**" package to add more extensive authentication system and **Facebook** login: https://www.wdtutorials.com/django/allauth-tutorial-facebook-login/.

17. Profile Model And Displayname

Let's create a "**Profile**" model so we can store more information about the user. There are other ways to do it but this is quite simple and less likely to cause complications.

Profile Model & OneToOneField

Edit "**models.py**" in *quarkblob/project/base/* and add these lines:

```
from django.contrib.auth.models import User
from django.db import models

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    display_name = models.CharField(default='', max_length=100)
```

"**user**" is defined as **OneToOneField**. *Profile* can link to only **one** user and *User* can link to only **one** *Profile*. Next we can add any additional fields like "**display_name**". This allows us to specify an *arbitrary* displayname that hides the actual *user account name*.

"**on_delete=models.CASCADE**" deletes the **profile** object if we delete the *referenced* **user** object. But it will not delete the user object if we delete the profile object. See more "*on_delete*" options: https://docs.djangoproject.com/en/dev/ref/models/fields/#django.db.mo

Displayname & Admin Inline Fields

In the *Admin* area we can allow users to edit **related objects** with *inline* classes. Let's use "**StackedInline**" class: https://docs.djangoproject.com/en/dev/ref/contrib/admin/#django.contr

Edit "**admin.py**" in *quarkblob/project/base/* and add these lines:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as
BaseUserAdmin
from django.contrib.auth.models import User

from base.models import Profile

class ProfileInline(admin.StackedInline):
    model = Profile
    can_delete = False
    verbose_name_plural = 'Profile'

class UserAdmin(BaseUserAdmin):
    inlines = (ProfileInline,)

admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

"**class ProfileInline(admin.StackedInline)**" defines an *inline admin descriptor*.

"**model = Profile**" defines the model we want to expose with the inline form.

"**can_delete = False**" defines if we can delete the related data from the inline form.

Without "**verbose_name_plural = 'Profile'**" we would see "*Profiles*" in the admin area.

"**class UserAdmin(BaseUserAdmin)**" defines a new user admin.
"**inlines = (ProfileInline,)**" adds the *profile inline* to the new admin.

At the end we have to **re-register** the "**UserAdmin**" for the "**User**" model.

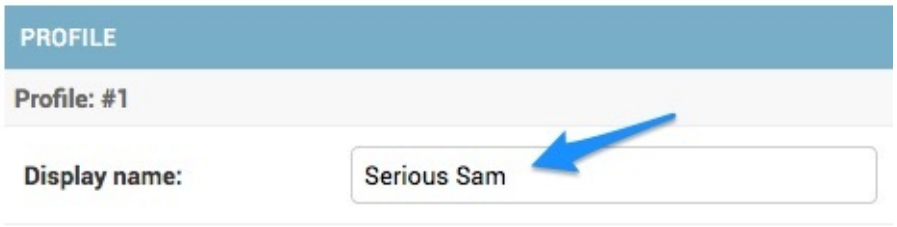
The admin page for managing user data works as it did before, but we also have an option to edit related profile data, like the **displayname**.

Run **migrations**:

```
python manage.py makemigrations && python manage.py migrate
```

Visit **admin** area and edit the superuser: **admin/auth/user/1/change/**.

Change the **displayname**:



Author Field & ForeignKey

"**Author**" will be defined as **ForeignKey**. That's a *many-to-one* relationship. *Author* can have **many** posts, but each post can have only **one** *Author*.

Edit "**models.py**" in *quarkblob/project/blog/*:

```
from django.contrib.auth.models import User
...

class Post(models.Model):
    ...
    tags = models.ManyToManyField(Tag, blank=True)
    author = models.ForeignKey(User, null=True, blank=True,
on_delete=models.\
CASCADE)
```

Run **migrations**:

```
python manage.py makemigrations && python manage.py migrate
```

Edit “**post.html**” in *quarkblob/project/blog/templates/blog/* and change the “**post-author**” element:

```
<span class="post-author">By <strong>admin</strong></span>
<span class="post-author">By <strong>
    {% if item.author.profile.display_name %}
        {{ item.author.profile.display_name }}
    {% else %}
    {{ item.author }}
    {% endif %}
</strong></span>
```

Update Custom Form

We haven’t actually stored **author** information for any posts. Let’s update the “**add_post**” view to set the current user as an author when we save the object.

Edit “**views.py**” in *quarkblob/project/blog/*:

```
def add_post(request):
    if request.method == "POST":
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            item = form.save(commit=False)
            item.author = request.user
            item.save()
    ...
```

Edit “**forms.py**” in *quarkblob/project/blog/* and **exclude** the author field:

```
        fields = '__all__'
        exclude = ['author']
```

Now users can’t change the author field when they are adding content. But we can always change the author using the admin area.

Login and visit “**blog/add/post/**” to add a post.

Now you will see user “**displayname**” in the *meta* section if it has been added to the profile. Otherwise we will display the **user account name**. Regular visitors can’t change the displayname yet but *admin* users can edit it in the admin area.

“**ANONYMOUS**” | This only works for new items. The old items will show **None** because there is no user linked to the objects. You could show **anonymous** here but you *probably* don’t want anonymous users to be able to add blog posts to your site. In the *permissions* chapter we restrict the access to the post form for logged-in users.

18. Profile Page

Profile Page will be a simple page to display information about the logged-in user and allow users to log-out.

Profile View

Edit "**views.py**" in *quarkblob/project/base/* and add a "**profile**" method:

```
def profile(request):
    return render(request, 'base/profile.html', {'title':
'Profile'})
```

Profile Template

Create "**profile.html**" in *quarkblob/project/base/templates/base/* and add these lines:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}
<div class="profile">

    <div class="profile-field">
        Username: <span class="profile-username"><strong>{{
request.user }}</\
strong></span>
    </div>

    {% if request.user.profile.display_name %}

        <div class="profile-field">
            Displayname: <span class="profile-email"><strong>{{
request.user.\
profile.display_name }}</strong></span>
```

```
        </div>

    {% endif %}

    <div class="profile-field">
        Email: <span class="profile-email"><strong>{{
request.user.email }}</\
strong></span>
    </div>

    <div class="profile-field">
        <a class="profile-logout submit" href="{% url 'logout'
%}">Logout</a>
    </div>

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

Profile Urls

Edit "**urls.py**" in *quarkblob/project/project/* and add the *profile* "**path**":

```
urlpatterns = [
    ...
    path('accounts/register/', views.Register.as_view(),
name='register'),
    path('accounts/profile/', views.profile, name='profile'),
    path('accounts/', include('django.contrib.auth.urls')),
]
```

Profile Styling

Create "**profile.scss**" in *quarkblob/project/base/static/base/css/*:

```
.profile {
  &-field {
    margin: 0.5em 0;
  }
}
```

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import “**profile**”:

```
@import "profile";
```

Visit **accounts/profile/** to see the **profile** page:

Profile Links

Edit “**base.html**” in *quarkblob/project/base/templates/base/*.

Add “**profile**” link to the **dropdown menu**:

```
{% if request.user.is_authenticated %}
    <li class="dropdown-li">
        <a class="dropdown-li-a" href="{% url 'profile' %}">{{
request.user.p\
rofile.display_name }}</a>
    </li>
    <li class="dropdown-li">
        <a class="dropdown-li-a" href="{% url 'logout'
%}">Logout</a>
    </li>
    ...
{% endif %}
```

Change the **top-right** corner link:

```
{% if not request.user.is_authenticated %}
    <a class="header-user show-mobile" href="{% url 'login'
%}">
        login / register
    </a>
{% endif %}
```

```
{% if request.user.is_authenticated %}
    <a class="header-user show-mobile" href="{% url 'profile'
%}">
        {{ request.user }}
    </a>
{% else %}
    <a class="header-user show-mobile" href="{% url 'login'
%}">
        login / register
    </a>
{% endif %}
```

Login & Logout Urls

Edit "**settings.py**" in *quarkblob/project/project/* and add these "**variables**":

```
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

Now when you **login** or **logout**, you will be redirected to the **homepage**.

19. Permissions And Decorators

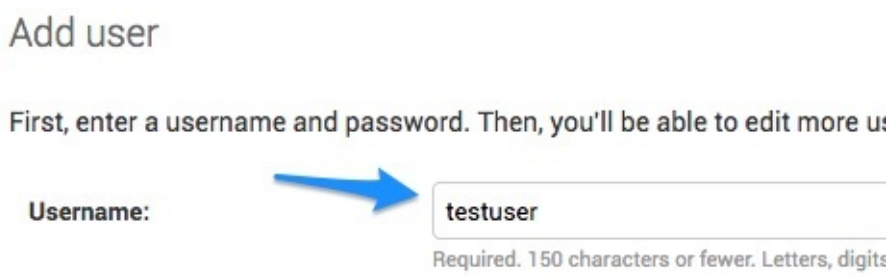
We can limit access to the **Post** form using *permissions* and *decorators*.

Editor Group

Visit *admin/auth/group/* and add a group called "**editor**".

Add "**Can add post**" and "**Can change post**" *permissions* to it:

Visit *admin/auth/user/add/* and add a "**testuser**".



Save and make the user member of the "**Editor**" group:

Edit "**post.html**" in *quarkblob/project/blog/templates/blog/* and add this *temporary* line to figure out what permissions the current user have:

```
<div class="post">
    <p>Current user permissions: {{
request.user.get_all_permissions }}</p>
    <span class="post-author">By <strong>
    ...
</div>
```

Open another browser and log in with "**testuser**". Visit some post page and you should see the permissions for the current user:

Now we can use these permissions to **restrict access**.

Decorators

> "**DECORATORS**" allow us to **dynamically alter** a function or class. Django provides some useful decorators related to *user access*:
> https://docs.djangoproject.com/en/dev/_modules/django/contrib/auth/decor

Edit “**views.py**” in *quarkblob/project/blog/* and add “**@permission_required**” *decorators*:

```
from django.contrib.auth.decorators import permission_required

@permission_required('blog.add_post')
def add_post(request):
   ...

@permission_required('blog.edit_post')
def edit_post(request, pk=None):
   ...
```

Edit “**views.py**” in *quarkblob/project/base/* and add “**@login_required**” *decorator*:

```
from django.contrib.auth.decorators import login_required

@login_required
def profile(request):
    return render(request, 'base/profile.html', {'title':
'Profile'})
```

Now if you **logout** and try to **add/edit** posts or visit the **/profile/** page, you will be redirected to the *login* page.

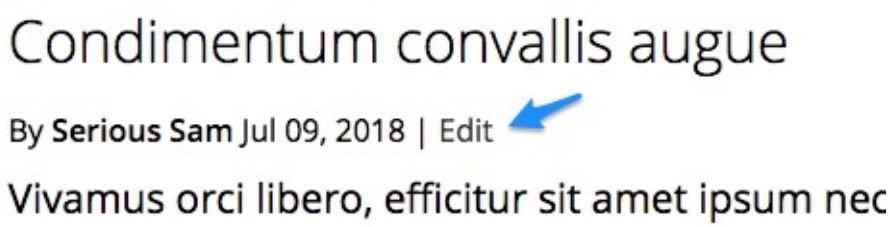
Edit Post Link

Edit “**post.html**” in *quarkblob/project/blog/templates/blog/* and remove the “**Current user permissions**” line and add a link to “**edit**” content:

```
<div class="post">
        <p>Current user permissions: {{
request.user.get_all_permissions }}</\
p>
        ...
        <span class="post-date">{{ item.post_date |date:"M d,
Y"}}</span>
        {% if perms.blog.change_post %}
                | <a class="post-edit" href="{% url 'edit_post'
```

```
item.id %}">E\
dit</a>
        {% endif %}
        <div class="post-description clearfix">{{
item.description }}</div>
        ...
    </div>
```

Now users with right permissions can see the “**Edit**” link and use it:



20. Extend User Forms

Let's add an option to manage "**email**" and "**displayname**" information with custom forms.

CustomUserCreationForm

Edit "**views.py**" in *quarkblob/project/base/* and change the "**Register**" class:

```
from base.forms import CustomUserCreationForm
class Register(generic.CreateView):
```

~~form_class = UserCreationForm~~

```
    form_class = CustomUserCreationForm     template_name =
'registration/register.html'

    success_url = reverse_lazy('login')
```

Django offers "**Generic**" views like *CreateView* for creating and editing content: https://docs.djangoproject.com/en/dev/ref/class-

based-views/generic-editing/.

Create *quarkblob/project/base/**forms.py*** and add "**CustomUserCreationForm**" class. The **bold** lines are changes to the original *UserCreationForm* class that I copied from here: "*quarkblob/venv/lib/python3.6/site-packages/django/contrib/auth/**forms.py***":

```
from django import forms


from django.contrib.auth.forms import UserCreationForm,
UsernameField from django.contrib.auth.models import User


from base.models import Profile


from django.utils.translation import gettext_lazy as _




class CustomUserCreationForm(UserCreationForm):




    email = forms.EmailField(
```

```
        label=_("Email"),

        help_text=_("Enter valid email address"),     )


    displayname = forms.CharField(

        label=_("Displayname"),         help_text=_("Enter
displayname."),         required=False,

    )


    class Meta:

        model = User

        fields = ("username",)

        fields = ("username", "email", "displayname",
"password1", "password2\

", )

        field_classes = {'username': UsernameField}
```

```
    def save(self, commit=True):

        user = super().save(commit=False)

        user.set_password(self.cleaned_data["password1"])
user.email = self.cleaned_data["email"]

        if commit:

            user.save()

            profile =
Profile(display_name=self.cleaned_data["displayname"])
profile.user = user             profile.save()

        return user
```

In here we subclass “**UserCreationForm**” to create “*CustomUserCreationForm*” and add “**email**” and “**displayname**” fields. We also save a new “**Profile**” object and relate that to the “**User**” object.

Now users can add *email* and *displayname* when registering to the site:

CustomUserChangeForm

Let's allow users to **edit** these fields.

Edit "**views.py**" in *quarkblob/project/base/* and add "**UserChange**" *class*:

```
from django.contrib.auth.models import User
from base.forms import CustomUserCreationForm,
CustomUserChangeForm
from django.contrib.auth.mixins import LoginRequiredMixin


class UserChange(LoginRequiredMixin, generic.UpdateView):
    model = User
    form_class = CustomUserChangeForm
    template_name = 'registration/user_change.html'
    success_url = reverse_lazy('login')
```

```
    def get_object(self):
        return self.request.user
```

> "**LoginRequiredMixin**" works with *class* views like the "**@login_required**" *decorator* with *function* based views: https://docs.djangoproject.com/en/dev/topics/auth/default/#the-loginrequired-mixin.

We add the "**get_object**" method so that the user will be always editing his own profile. It doesn't matter what user **id** is sent to the view in the url.

Edit "**forms.py**" in *quarkblob/project/base/* and add "**CustomUserChangeForm**":

```
from django.contrib.auth.forms import UserCreationForm,
UserChangeForm, ReadO\

nlyPasswordHashField, UsernameField
class CustomUserChangeForm(UserChangeForm):

    email = forms.EmailField(
```

```
        label=_("Email"),

        help_text=_("Enter valid email address"),     )


    displayname = forms.CharField(

        label=_("Displayname"),         help_text=_("Enter
displayname."),         required=False,

    )




    password = ReadOnlyPasswordHashField(

        label=_("Password"),

        help_text=_(

            "Raw passwords are not stored, so there is no way
to see this "

            "user's password, but you can change the password
using "

            "<a href=\"/accounts/password_change/\">this
form</a>."
```

```
        ),

    )


    class Meta:

        model = User

        fields = '__all__'

        fields = ("username", "email", "displayname",
"password")         field_classes = {'username': UsernameField}


    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)

        self.profile = Profile.objects.get(user=self.instance)
self.fields['displayname'].initial = self.profile.display_name
f = self.fields.get('user_permissions')

        if f is not None:
```

```
            f.queryset =
f.queryset.select_related('content_type')


    def save(self, commit=True):


        user = super().save(commit=False)         user.email =
self.cleaned_data["email"]





        if commit:


            user.save()


            self.profile.display_name =
self.cleaned_data["displayname"]


            self.profile.save()         return user





    def clean_password(self):


        return self.initial["password"]
```

The original “**UserChangeForm**” *class* was also copied from here “*quarkblob/venv/lib/python3.6/site-packages/django/contrib/auth/**forms.py***”.

Edit “**urls.py**” in *quarkblob/project/project/* and add “**user_change**” path:

```
urlpatterns = [

    ...

    path('accounts/register/', views.Register.as_view(),
name='register'),     path('accounts/change/<int:pk>/',
views.UserChange.as_view(), name='user_\

change'),

    path('accounts/profile/', views.profile, name='profile'),
...

] + static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

> **Note**: the “*user_change*” path is expecting an **int** as a *parameter*, but we are overriding it using the “**get_object()**” method in the **UserChange** class to get the “**current user**” id.

Create “**user_change.html**” in *quarkblob/project/templates/registration/*:

```
{% extends 'base/base.html' %}

{% block title %}Edit Profile{% endblock %}

{% block header %}Edit Profile{% endblock %}

{% block content %}

  <form method="post">
    {% csrf_token %}

    {{ form.as_p }}

    <button class="submit" type="submit">Change</button>
  </form>
{% endblock %}
```

Logged-in user can now visit “**/accounts/change/[some_int]/**” to manage profile information:

Username:

admin

Required. 150 characters or fewer. Letters, digits and @/./+/-

Email:

myemail@gmail.com

Enter valid email address

Displayname:

Serious Sam

Enter displayname.

Profile Page Links

Edit "**profile.html**" in *quarkblob/project/base/templates/base/* and "**Change password**" and "**Edit profile**" links:

```
<div class="profile-field">     <a class="profile-edit submit"
href="{% url 'user_change' request.user.pk\


 %}">Edit profile</a>     <a class="profile-change-password
submit" href="{% url 'password_change' \
```

```
%}">Change password</a>     <a class="profile-logout submit"
href="{% url 'logout' %}">Logout</a> </div>
```

Visit “**/accounts/profile/**” and you can see the new **links**:

21. Test Driven Development

In this chapter we create a "**Feedback**" feature using **Test Driven Development** process.

Imagine building a complex site that you have to extend or update regularly. It would be really painful and unreliable to **manually** test if the changes brake something. But if you have written *correct* tests for **everything**, then you can be lot more confident that the software works after the changes.

This can also provide you a "**solid process**" to build applications. You will be thinking software in terms of the **end user**, not in terms of what you *happen* to build. It's harder to build unnecessary things if you first have to justify them in **user stories**.

> "**USER STORY**" is an **informal description** of a feature or features. They are often written from the perspective of the end user.

At first it can be difficult to figure out "**what**" and "**how**" to test. Testing *some* parts of the software is better than testing *nothing*. Just **try** it and you will get better.

Functional Tests VS Unit Tests

Here is how the process goes in general:

- Write a "**functional test**" to test the **end user** experience. For example in our first test we test if the user sees the text

"**Feedback**" in the title when she visits a page **/feedback**/. We use **Selenium** to actually open a real web browser and check if the title contains that word.
- After the test fails, we need to write **code** so that the user can see the *Feedback* text. Whenever we are about to write code, we first write "**unit tests**" for the code.
- When the *unit tests* and the *functional test* **passes**, we have fulfilled a certain requirement ("*User sees the text Feedback in /feedback/ page.*"). Then we can move to the next functional test and repeat.

> "**SELENIUM**" is a **testing framework** for web apps: https://www.seleniumhq.org/.

"**Functional Tests**" goes in *quarkblob/project/**functional_tests**/* because they can test functionality that spans *multiple apps*.

"**Unit Tests**" we will put inside *app* folders: *quarkblob/project/feedback/tests/**test_feedback_views.py***.

Selenium & Geckodriver

Selenium needs a "**driver**" (http://selenium-python.readthedocs.io/installation.html#drivers) to interact with the browser. I'm using **Firefox** and **Geckodriver**: https://github.com/mozilla/geckodriver.

In OSX you can do this to install the driver:

```
brew install geckodriver
```

> "**MANUAL INSTALLATION**" | You can install it manually for **Windows**, **Linux** and **OSX** by downloading it from here: https://github.com/mozilla/geckodriver/releases. Extract the package and put the geckodriver file in the system **PATH** like "**/usr/local/bin**". Make it an executable for unix systems: "**chmod +x geckodriver**".

If you have trouble with *Firefox*, try "**Chrome**": https://sites.google.com/a/chromium.org/chromedriver/downloads. Just use "*from selenium.webdriver.**chrome**.webdriver import WebDriver*" instead of "*from selenium.webdriver.**firefox**.webdriver import WebDriver*".

Install **selenium** package:

```
pip install selenium
```

Functional Tests

Create "**test_feedback_page.py**" in *quarkblob/project/**functional_tests**/*.

Here are the "**User stories**" that describe the user interactions:

```
# She sees "Feedback" in the title and header
# She sees a list of feedback items
# She sees an input form to leave feedback
# She fills in the form and sees her feedback appear
```

Let's write test for the first *user story*:

```
import time
from django.contrib.staticfiles.testing import
StaticLiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver
```

```
class FeedbackTestBase(StaticLiveServerTestCase):

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.browser = WebDriver()

    @classmethod
    def tearDownClass(cls):
        cls.browser.quit()
        super().tearDownClass()

class FeedbackPageTests(FeedbackTestBase):

    def test_feedback_page(self):

        # She sees "Feedback" in the title and header

        self.browser.get('%s%s' % (self.live_server_url,
'/feedback/'))
        time.sleep(5)
        self.assertIn('Feedback', self.browser.title)

        # She sees a list of feedback items
        # She sees an input form to leave feedback
        # She fills in the form and sees her feedback appear
```

First we create a "**FeedbackTestBase**" by subclassing *StaticLiveServerTestCase*. This allows us to inherit from the base so we don't have to repeat ourself too much.

"**StaticLiveServerTestCase**" launches a **live Django server** in the background. This allows us to use **Selenium** to execute **functional tests** inside the browser.

> "**StaticLiveServerTestCase**" doesn't use the development server or database but runs an isolated setup everytime we run the tests.

“**setUpClass**” is called before the tests. It must be decorated with “**@classmethod**”.

“**tearDownClass**” is called after the tests. It must be also decorated with “**@classmethod**”.

“**test_feedback_page**” method is where we put the actual **tests**.

“**self.browser.get(‘%s%s’ % (self.live_server_url, ‘/feedback/’))**” will visit the test server url.

“**time.sleep(5)**” will wait approximately 5 seconds.

“**self.assertIn(‘Feedback’, self.browser.title)**” checks if the page “**<title>**” element contains the string “**Feedback**”.

Run tests:

```
python manage.py test functional_tests
```

You should now see **browser** opening and an “**AssertionError**” in the terminal:

```
... AssertionError: 'Feedback' not found in ''
```

The test **failed** as it should.

Unit Tests

Create a new app called **Feedback**:

```
python manage.py startapp feedback
```

Edit “**settings.py**” in *quarkblob/project/project/* and add “**feedback**” to the “**INSTALLED_APPS**” list:

```
INSTALLED_APPS = [
    ...
    'feedback',
]
```

Delete "**tests.py**" in */quarkblob/project/feedback/*.

Create "**__init__.py**" in *quarkblob/project/feedback/**tests**/*".

> "**__INIT__.PY**" makes Python treat this directory as a **package**: https://docs.python.org/3/tutorial/modules.html#packages.

Create "**test_feedback_views.py**" in *quarkblob/project/feedback/tests/* and add these lines:

```
from django.test import TestCase, Client

class BaseTest(TestCase):

    def setUp(self):
        self.client = Client()

class FeedbackViewTest(BaseTest):

    def test_uses_right_template(self):
        response = self.client.get('/feedback/')
        self.assertTemplateUsed(response,
'feedback/feedback.html')
```

> "**UNITTEST**" is a **Unit testing framework** https://docs.python.org/3/library/unittest.html#module-unittest.

We subclass the *TestCase* to setup a "**BaseTest**" class.

“**Client**” class acts as a *dummy* “Web browser”. This allows us interact with the application programmatically.

> “**UNIT TESTS ARE FASTER**” | Running **unit tests** is faster than **functional tests**. With **functional tests** we actually open a real browser, but with **unit tests** we access the app *programmatically*.

“**test_uses_right_template**” method tests if the right **template** is used when we visit **/feedback/**. Run the tests:

```
python manage.py test feedback
```

You should get an *AssertionError*:

```
... AssertionError: No templates used to render the response
```

Edit “**views.py**” in *quarkblob/project/feedback/* and fill it with these lines:

```
from django.shortcuts import render

def feedback(request):

    return render(request, 'feedback/feedback.html', {'title':
'Feedback'})
```

Create “**feedback.html**” in *feedback/**templates/feedback/***:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}{% endblock %}
```

Edit “**urls.py**” in *quarkblob/project/project/* and include “**feedback.urls**”:

```
urlpatterns = [
    ...
    path('accounts/', include('django.contrib.auth.urls')),
    path('feedback/', include('feedback.urls')),
    path('', include('base.urls'))
] + static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

Create "**urls.py**" in *quarkblob/project/feedback/*:

```
from django.urls import path
from feedback import views

urlpatterns = [
    path('', views.feedback, name='feedback'),
]
```

Run **unit tests** again:

```
python manage.py test feedback
```

It should now **pass**:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.029s

OK
Destroying test database for alias 'default'...
```

Also **functional tests** should now pass:

```
python manage.py test functional_tests
```

Let's finnish the test by checking the "**<h1>**" element as well.

Edit "**test_feedback_page.py**" in *quarkblob/project/functional_tests/*:

```
# She sees "Feedback" in the title and header

        self.browser.get('%s%s' % (self.live_server_url,
'/feedback/'))
        time.sleep(5)
        self.assertIn('Feedback', self.browser.title)

        h1 = self.browser.find_element_by_tag_name('h1').text
        self.assertEqual('Feedback', h1)
        time.sleep(5)
```

"**h1 = self.browser.find_element_by_tag_name('h1').text**" finds the "**<h1>**" element and stores its content in the **h1** variable.

You can test if this **really** works by changing the "**Feedback**" title. Let's do that.

Edit "**views.py**" in *quarkblob/project/feedback/* and change the title:

```
from django.shortcuts import render

def feedback(request):
    return render(request, 'feedback/feedback.html', {'title':
'Feedback'})
    return render(request, 'feedback/feedback.html', {'title':
'FeedbackTEST'\
})
```

Run **functional tests**:

```
python manage.py test functional_tests
```

Now the test fails:

```
self.assertEqual('Feedback', h1)
AssertionError: 'Feedback' != 'FeedbackTEST'
- Feedback
+ FeedbackTEST
?         ++++
```

Change the title back to "**Feedback**".

At this point both of these tests should **pass**:

```
python manage.py test feedback
python manage.py test functional_tests
```

This means that we have succesfully written **tested functionality** to satisfy this requirement: "**She sees "Feedback" in the title and h1 tags**".

You can of course visit "**/feedback/**" manually to see if the requirement is fulfilled for the *actual* site as well.

Feedback Items

Let's tackle the next user story: "**She sees a list of feedback items**".

Edit **test_feedback_page.py** in *quarkblob/project/functional_tests/* and do the following changes:

```
from feedback.models import Feedback

class FeedbackTestBase(StaticLiveServerTestCase):

    @classmethod
    def setUpClass(cls):
        Feedback.objects.create(title='Best blog ever!')
        Feedback.objects.create(title='Django rules!')
        super().setUpClass()
        cls.browser = WebDriver()

    @classmethod
    def tearDownClass(cls):
        cls.browser.quit()
        super().tearDownClass()

class FeedbackPageTests(FeedbackTestBase):

    def test_feedback_page(self):
```

```
        # She sees "Feedback" in the title and header

        ...

        # She sees a list of feedback items

        feedback_items =
self.browser.find_element_by_class_name('feedback-it\
ems').text
        self.assertIn('Best blog ever!', feedback_items)
        self.assertIn('Django rules!', feedback_items)

        time.sleep(5)

        # She sees an input form to leave feedback
        # She fills in the form and sees her feedback appear
```

First we try to create 2 feedback **objects** in the "**setUpClass**" method.

Then we search for the object titles inside an element with a class name "**feedback-items**".

Run the **functional test**…

```
python manage.py test functional_tests
```

…and it will **fail**:

```
... ImportError: cannot import name 'Feedback'
```

We are trying to "**import Feedback**", create **objects** and see the object titles on the page.

But we don't have the **Feedback** model yet. For that to exist we have to write **code**. And before we write any code we write a **unit test** for the code.

Testing Models

Create “**test_feedback_models.py**” in *quarkblob/project/feedback/tests/*:

```
from django.test import TestCase

from feedback.models import Feedback

class FeedbackModelTest(TestCase):

    def setUp(self):

        item = Feedback.objects.create(title='Test title')

        self.item = item

    def test_saving_items(self):
        self.assertEqual(self.item.title, 'Test title')

    def test_str(self):
        self.assertEqual('Test title', self.item.__str__())

    def test_absolute_url(self):
        self.assertEqual('/feedback/test-title/',
self.item.get_absolute_url(\
))
```

First we try to import **Feedback** and create an object. Then we test if the saved item has the right “**title**” and if the “**text representation**” of the object is correct. And finally we test if “**get_absolute_url**” returns the correct path.

> “**DUPLICATE CODE**” | Sometimes it might seem that you are testing same things in both test types and indeed the tests can **overlap**. That’s ok. But these different test types test different things. “**unit tests**” are tests for the *internal workings* and “**functional tests**” for the *end user experienc*”. Try to keep tests *isolated* and not to overthink it.

Run the unit tests:

```
python manage.py test feedback
```

Test will *fail*:

```
... ImportError: cannot import name 'Feedback'
```

You can **limit** the tests to be run like this:

```
python manage.py test feedback.tests.test_feedback_models
```

Create "**models.py**" in *quarkblob/project/feedback/*:

```
from django.db import models
from django.utils.text import slugify
from django.urls import reverse

class Feedback(models.Model):

    title = models.CharField(max_length=255, default='')
    slug = models.SlugField(default='', blank=True)

    def __str__(self):
        return '%s' % self.title

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super().save(*args, **kwargs)

    def get_absolute_url(self):
        return reverse('feedback_item', args=[str(self.slug)])
```

Run **migrations**:

```
python manage.py makemigrations && python manage.py migrate
```

> "**SMALL STEPS**" | You can write more code at once and *not* take this many steps. But it might help you understand the process better

when you take smaller steps.

Running "**python manage.py test feedback.tests.test_feedback_models**" gives us this:

```
... 'feedback_item' is not a valid view function or pattern
name.
```

Edit "**urls.py**" in *quarkblob/project/feedback/* and add path for "**feedback_item**":

```
urlpatterns = [
    path('', views.feedback, name='feedback'),
    path('<slug:slug>/', views.feedback_item,
name='feedback_item'),
]
```

Running "**python manage.py test feedback.tests.test_feedback_models**" gives us this:

```
... 'feedback.views' has no attribute 'feedback_item'
```

Edit "**views.py**" in *quarkblob/project/feedback/* and add "**feedback_item**" method:

```
from django.shortcuts import render, get_object_or_404
from feedback.models import Feedback

def feedback_item(request, slug=None):

    _item = get_object_or_404(Feedback, slug=slug)

    return render(request, 'feedback/feedback_item.html',
{'title': _item})
```

Create "**feedback_item.html**" in *feedback/templates/feedback/*:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}
<div class="feedback-item"></div>
{% endblock %}
```

> "**FEEDBACK ITEM PAGE**" | We are not actually going to write tests for the "**feedback/<slug>/**" page because it doesn't add anything *new* to the tests. And more importantly, we don't have a "**user story**" for that feature. This is just a *placeholder* template for possible future functionality.

"**python manage.py test feedback.tests.test_feedback_models**" will now "**PASS**".

This means that we can now "**create**" feedback objects. Also *Feedback* class "**__str__**" and "**get_absolute_url**" methods are working correctly.

Running "**python manage.py test functional_tests**" will now get us a bit further:

```
... Unable to locate element: .feedback-items
```

Testing Views

Let's test if the **Feedback** view passes the right **"feedback elements"** to the view.

Edit **test_feedback_views.py** in *quarkblob/project/feedback/tests/* folder.

This is how it should look after adding the "**marked lines**":

```
from django.test import TestCase, Client
from feedback.models import Feedback

class BaseTest(TestCase):

    def setUp(self):
        Feedback.objects.create(title='title1')
        Feedback.objects.create(title='title2')

        self.client = Client()

class FeedbackViewTest(BaseTest):
    client = Client()

    def test_uses_right_template(self):
        response = self.client.get('/feedback/')
        self.assertTemplateUsed(response,
'feedback/feedback.html')

    def test_passes_right_items(self):
        response = self.client.get('/feedback/')
        items = Feedback.objects.all()
        for response_item in response.context['items']:
            self.assertIn(response_item.title, [item.title for
item in items])
```

All added items should be found in "**response.context['items']**". "**test_passes_right_items**" will go through all the items and compare them to the items in the database.

Running "**python manage.py test feedback.tests.test_feedback_views**" will now give us this:

```
... KeyError: 'items'
```

Edit "**views.py**" in *quarkblob/project/feedback/* and pass "**Feedback**" *objects* to the view:

```
def feedback(request):
    items = Feedback.objects.all()

    return render(request, 'feedback/feedback.html', {'title':
```

```
~~'Feedback'})~~
    return render(request, 'feedback/feedback.html', {'items':
items', 'title\
': 'Feedback',})
```

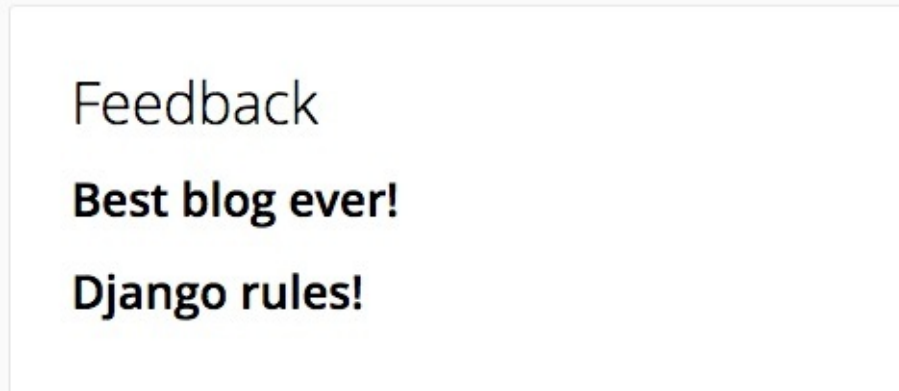
"**python manage.py test feedback.tests.test_feedback_views**" should **PASS**. We are now passing the right objects to the view.

Now we only need to loop through the feedback objects and print them out in the template to make the **functional tests** pass.

Edit "**feedback.html**" in *feedback/templates/feedback/* and add "**feedback-items**" element:

```
{% extends 'base/base.html' %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}
    <div class="feedback-items">
        {% for item in items %}
            <h2 class="feedback-item">{{ item }}</h2>
        {% endfor %}
    </div>
{% endblock %}
```

Run "**python manage.py test functional_tests**" and you should see this while the test browser window is open:



Now we have succesfully written **tested functionality** to satisfy the requirement: "**She sees a list of feedback items**".

Testing Forms

Let's tackle the last requirement next: "**She sees an input form to leave feedback, fills in the form and sees her feedback appear**".

Edit *project/functional_tests/**test_feedback_page.py***

```
...
from selenium.webdriver.common.keys import Keys

class FeedbackPageTests(FeedbackTestBase):

    def test_feedback_page(self):

        # She sees "Feedback" in the title and header

        ...

        time.sleep(5)

        # She sees a list of feedback items

        ...

        # She sees an input form to leave feedback, fills in
the form and see\
s her feedback appear

        input = self.browser.find_element_by_id('id_title')
        time.sleep(2)
        input.send_keys('Awesome blog!')
        input.send_keys(Keys.ENTER)
        time.sleep(5)
        feedback_items =
self.browser.find_element_by_class_name('feedback-it\
ems').text
        self.assertIn('Awesome blog!', feedback_items)
```

“**input.send_keys(‘Awesome blog!’)**” and “**input.send_keys(Keys.ENTER)**” will fill in the input and submit the form.

Run “**python manage.py test functional_tests**” and you should see this:

```
... Unable to locate element: [id="id_title"]
```

> We **don’t** have to test how the Django forms work in general because we haven’t coded that functionality ourselves. Only test your “**own code**”.

Let’s add the form functionality.

Create *quarkblob/project/feedback/**forms.py***:

```
from django.forms import ModelForm
from .models import Feedback

class FeedbackForm(ModelForm):
    class Meta:
        model = Feedback
        fields = {'title'}
```

Edit *quarkblob/project/feedback/**views.py*** and add “**form handling**” for the *feedback* view:

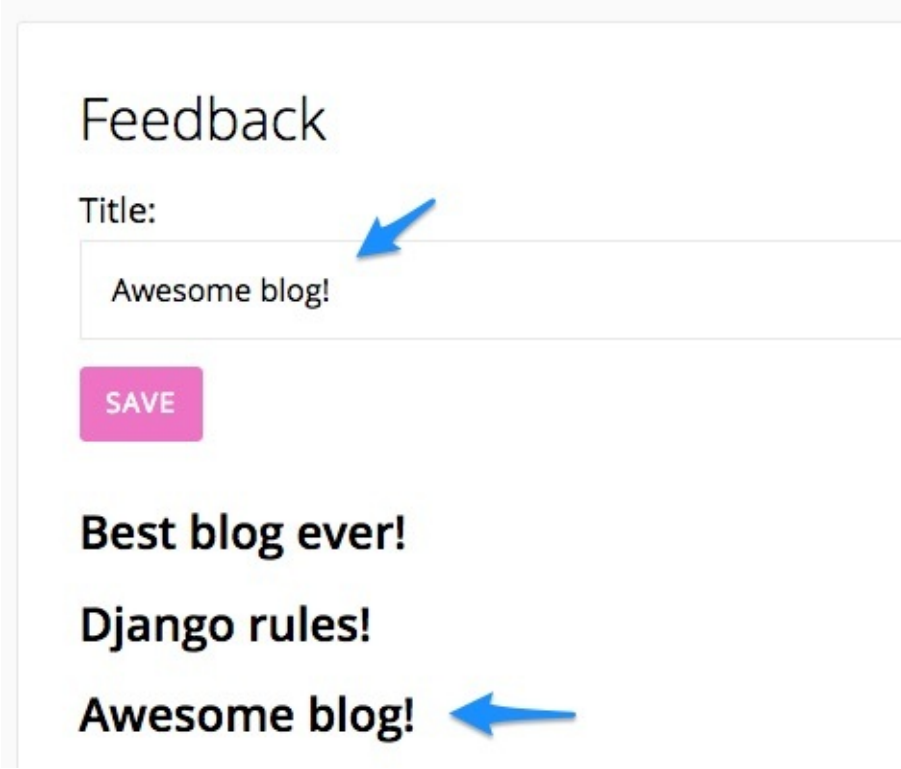
```
from feedback.forms import FeedbackForm

def feedback(request):
    items = Feedback.objects.all()
    if request.method == "POST":
        form = FeedbackForm(request.POST)
        if form.is_valid():
            item = form.save(commit=False)
```

```
            item.save()
    else:
        form = FeedbackForm()

    return render(request, 'feedback/feedback.html', {'items': items, 'title'\
: 'Feedback'})
    return render(request, 'feedback/feedback.html', {'items':
items, 'form':\
 form, 'title': 'Feedback',})
```

Edit **feedback.html** in *quarkblob/project/feedback/templates/feedback/* and add the *form* "**markup**":

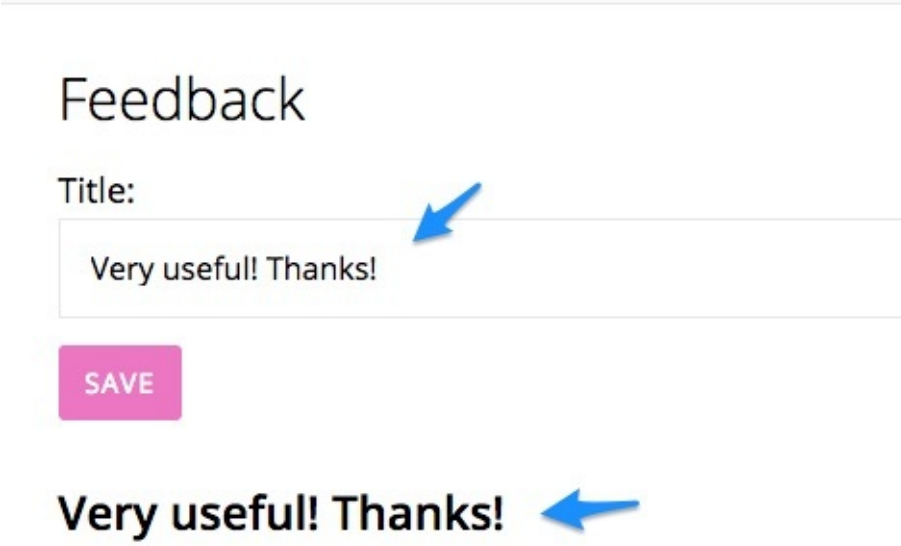
```
% block content %}
    <form class="feedback-form" method="POST" action="">
        {% csrf_token %}
        {{ form }}
        <input class="feedback-form-submit submit"
type="submit" value="Save">
    </form>

    <div class="feedback-items">
        ...
    </div>
{% endblock %}
```

Run "**python manage.py test functional_tests**" and you should see selenium **submitting** the form and the title **appearing** at the bottom:



Now the last requirement "**She sees an input form to leave feedback, fills in the form and sees her feedback appear**" is also fulfilled.

You can visit "**/feedback/**" to test it manually:



22. Translation

In this chapter you will learn how to start creating "**multilingual**" websites.

i18n_patterns

Edit "**settings.py**" in *quarkblob/project/project/*, add "**LocaleMiddleware**", "**LANGUAGES**" list and "**LOCALE_PATHS**" list:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]

LANGUAGE_CODE = 'en-us'

LANGUAGES = [
    ('en', 'English'),
    ('de', 'German'),
]

LOCALE_PATHS = [
    os.path.join(BASE_DIR, 'locale'),
]
```

Edit "**urls.py**" in *quarkblob/project/project/* and replace the content with this:

```
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include
from django.conf.urls.i18n import i18n_patterns
```

```
from django.utils.translation import gettext_lazy as _

from base import views
from project import settings

urlpatterns = [
    path('i18n/', include('django.conf.urls.i18n')),
    path('admin/', admin.site.urls),
    path('accounts/register/', views.Register.as_view(),
name='register'),
    path('accounts/change/<int:pk>/',
views.UserChange.as_view(), name='user_\
change'),
    path('accounts/profile/', views.profile, name='profile'),
    path('accounts/', include('django.contrib.auth.urls')),
]

urlpatterns += i18n_patterns(
    path('', include('base.urls')),
    path('blog/', include('blog.urls')),
    path('tags/<slug:slug>/', views.tag, name='tag'),
    path('feedback/', include('feedback.urls')),
    path('search/', include('search.urls')),
    path(_('about/'), views.about, name='about'),
    prefix_default_language=False,
) + static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

First we put **non-translatable** paths inside the "**urlpatterns**" list. Then we add the paths we want to enable translation with inside "**i18n_patterns()**" function and *append* those to the **urlpatterns** list.

> "**gettext()**" function specifies a string to be **translated**.

It is useful to create a **shorthand** for the "**gettext_lazy()**" function: "**from django.utils.translation import gettext_lazy as _**".

With "**_('about/')**" we mark the path to be translated.

Now we can translate “**/about/**” path and its content.

About Page

Edit “**views.py**” in *quarkblob/project/base/* and add “**about**” view:

```
from django.utils.translation import gettext as _
def about(request):
    return render(request, 'base/about.html', {'title':
_('About')})
```

Create “**about.html**” in *quarkblob/project/base/templates/base/*:

```
{% extends 'base/base.html' %}
{% load i18n %}
{% block title %}{{ title }}{% endblock %}
{% block h1 %}<h1>{{ title }}</h1>{% endblock %}
{% block content %}
<div class="about">
    <p>{% trans 'I want to translate this paragraph.' %}</p>
</div>
{% endblock %}
```

> Put “**{% load i18n %}**” towards the top so the template can use *translation* tags. You have to use this in the child template even if the *parent* template has it. “**{% trans ‘YOURSTRING’ %}**” tag translates the string if there is a translation string available for the user’s language.

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and change “**about**” menu links:

```
<li class="menu-li">
    <a class="menu-li-a {% url 'about' as about_url %} {% if
request.get_full\
_path == about_url %} menu-li-a--active{% endif %}" href="{%
url 'about' %}">\
```

```
{% trans 'About' %}</a>
</li>
...
<li class="dropdown-li">
    <a class="dropdown-li-a" href="{% url 'about' %}">{% trans
'About' %}</a>
</li>
```

> “**URL TAG**” | Great thing about using **url** tag like this “**{% url ‘about’ %}**” is that it returns the **translated** path.

Makemessages & Compilemessages

Create “**locale**” folder in *quarkblob/project/*.

Make and **compile** translation files:

```
django-admin makemessages -l de && django-admin compilemessages
-l de
```

> “**MESSAGES**” | You don’t have to run both commands at the same time if you are just *making* the message files based on the translation markers in the templates or *compiling* the messages when you add the translations in the **.po** files but I run them at the same time for convenience.

Edit “**django.po**” in *quarkblob/project/locale/de/LC_MESSAGES/* and add “**translations**”:

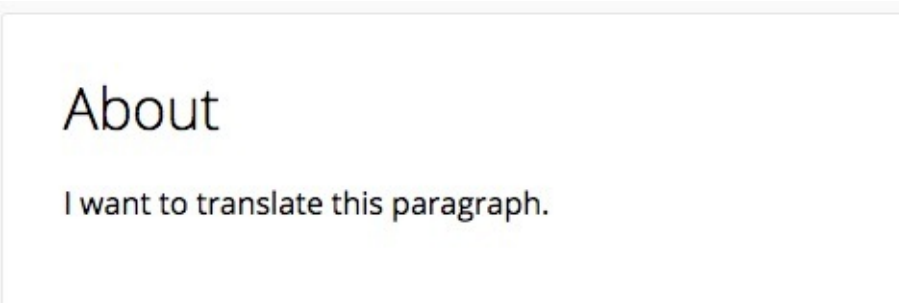
```
#: base/templates/base/about.html:7
msgid "I want to translate this paragraph."
msgstr "Ich mochte...."
```

```
#: base/templates/base/base.html:27
base/templates/base/base.html:39
#: base/views.py:44
msgid "About"
msgstr "Uber uns"

#: project/urls.py:31
msgid "about/"
msgstr "uber-uns/"
```

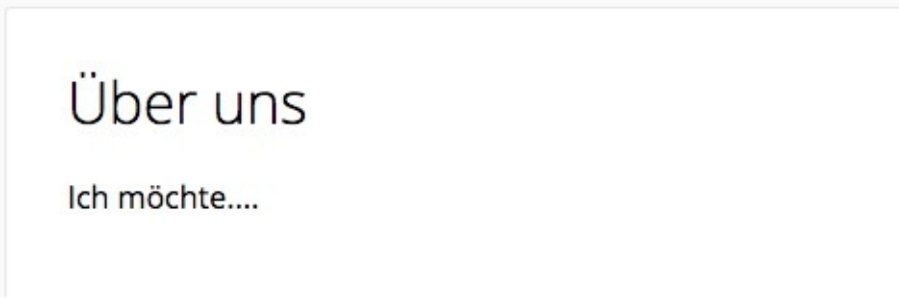
Make and **compile** translation files:

```
django-admin makemessages -l de && django-admin compilemessages
-l de
```

Now if you visit "**/about/**", you will see this:



And this in "**/de/uber-uns/**":



Language Switcher

I'm using the code from here
https://docs.djangoproject.com/en/dev/topics/i18n/translation/#the-set-

language-redirect-view to create a “**language switcher**”.

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and add the “**language-switcher**” form:

```
<div class="footer">
    <form class="language-switcher" action="{% url
'set_language' %}" method=\
"post">{% csrf_token %}
        <input name="next" type="hidden" value="/" >
        <select name="language">
            {% get_current_language as LANGUAGE_CODE %}
            {% get_available_languages as LANGUAGES %}
            {% get_language_info_list for LANGUAGES as
languages %}
            {% for language in languages %}
                <option value="{{ language.code }}"{% if
language.code == LAN\
GUAGE_CODE %} selected{% endif %}>
                    {{ language.name_local }} ({{ language.code
}})
                </option>
            {% endfor %}
        </select>
        <button type="submit">Go</button>
    </form>
</div>
```

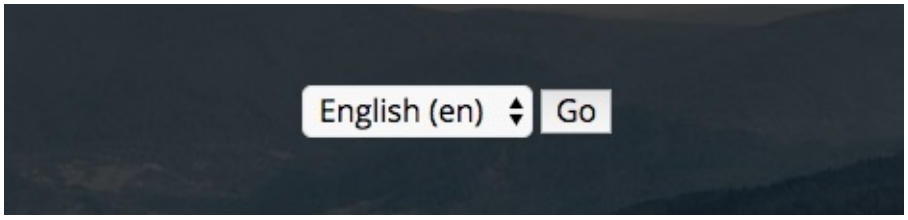
Create “**language_switcher.scss**” in *quarkblob/project/base/static/base/css/*:

```
.language-switcher {
  select, &-submit{
    width: auto;
    padding: 0;
  }
}
```

Edit “**style.scss**” in *quarkblob/project/base/static/base/css/* and import “**language_switcher**”:

```
@import "language_switcher";
```

And there we have it:



23. Custom Error Pages

You might want to create **custom** error pages. You can do this easily by adding templates in the *root* "**templates**" directory.

Settings

Edit "**settings.py**" in *quarkblob/project/project/* and setup "**DEBUG**" and "**ALLOWED HOSTS**" variables:

```
DEBUG = False
ALLOWED_HOSTS = ['localhost']
```

For testing purposes we need to *temporarily* setup "**DEBUG = False**" for the custom templates to kick in.

Template Files

Create the templates:

- *quarkblob/project/templates/**e_base.html***
- *quarkblob/project/templates/**400.html***
- *quarkblob/project/templates/**403.html***
- *quarkblob/project/templates/**404.html***
- *quarkblob/project/templates/**500.html***

"**e_base.html**":

```
<!doctype html>
{% load i18n %}
<html>
<head>
    <meta charset="utf-8">
```

```
    <meta name="viewport" content="width=device-width, user-
scalable=no, init\
ial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <link href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400,600\
" rel="stylesheet">
    <style>
        .error {
            font-family: "Open Sans", sans-serif;
            font-size: 1.8em;
            padding-top: 2em;
            text-align: center;
            line-height: 2em;
            color: #333;
        }
        .error-message-error {
            color: #666;
            font-size: 0.8em;
            font-weight: 300;
        }
        img { border-radius: 3px; width: 300px; }
        a { color: #666; }
    </style>
    <title>{% block error_message %}{% endblock %}</title>
</head>
<body class="error-page">
<div class="error">
    <div class="error-brand"><a class="error-brand-a"
href="/">QuarkBlob.com<\
/a></div>
    <div class="error-message">{% block error_title %}{%
endblock %}</div>
</div>
</body>
</html>
```

“400.html”:

```
{% extends "e_base.html" %}
{% load i18n %}

{% block error_title %}
    {% trans "Sorry, we can't seem to be able to process your
request at the \
moment." %}
    <div class="error-message-error">{% trans "Bad Request
```

```
(400)" %}</div>
{% endblock %}
{% block error_message %}{% trans "Bad Request (400)" %}{%
endblock %}
```

“403.html”:

```
{% extends "e_base.html" %}
{% load i18n %}

{% block error_title %}
    {% trans "Sorry, you don't have permissions to access this
page." %}
    <div class="error-message-error">{% trans "Permission
Denied (403)" %}</d\
iv>
{% endblock %}
{% block error_message %}{% trans "Permission Denied (403)" %}
{% endblock %}
```

“404.html”:

```
{% extends "e_base.html" %}
{% load i18n %}

{% block error_title %}
    {% trans "Sorry, we can't seem to find the page you are
looking for." %}
    <div class="error-message-error">{% trans "Page Not Found
(404)" %}</div>
{% endblock %}
{% block error_message %}{% trans "Page Not Found (404)" %}{%
endblock %}
```

“500.html”:

```
{% extends "e_base.html" %}
{% load i18n %}

{% block error_title %}
    {% trans "Sorry, the service is not fully functional at the
moment." %}
    <div class="error-message-error">{% trans "Server Error
```

```
(500)" %}</div>
{% endblock %}
{% block error_message %}{% trans "Server Error (500)" %}{%
endblock %}
```

Visit “**http://localhost:8000/something**” and you should see the custom “**page not found**” page.

Read more: https://docs.djangoproject.com/en/dev/ref/views/#error-views

24. Deployment

Setting up **servers** can be done in many ways. In here I present *one* way of doing it with **Digitalocean droplet**, **Bitbucket**, **Nginx**, **Gunicorn** and **Supervisor**. We will also setup a **domain**, **HTTPS**, **mailbox** and **caching**. I try to keep this *concise* but it is highly recommended to have a "**deeper knowledge**" of all the underlying technologies.

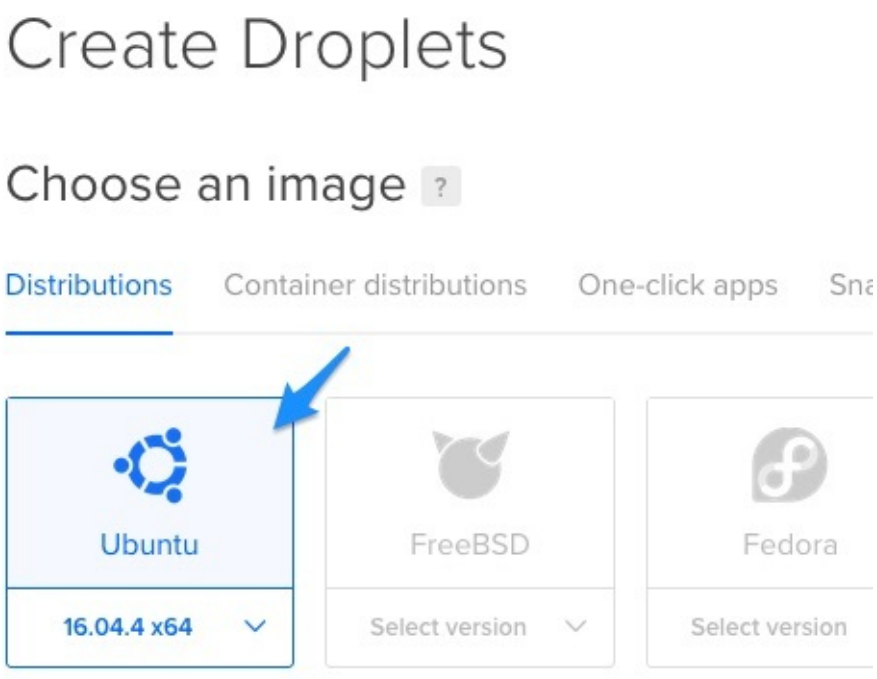
You can also *automate* **most** of this with tools like https://www.ansible.com/, https://www.chef.io/chef/ and https://get.fabric.io/.

Digitalocean Droplet

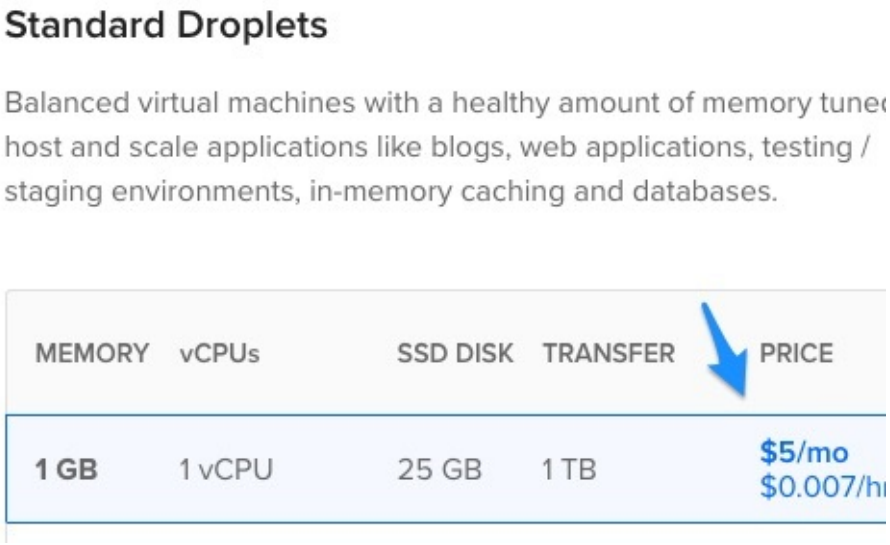
Digitalocean is a cheap way to test deployment because they charge *by the hour*. For example the cheapest droplet they offer is at the moment under 10 bucks / month. If you use it for couple of days, it costs you practically nothing.

Go to http://www.digitalocean.com and buy a "**droplet**":

I'm using the default "**Ubuntu image**":

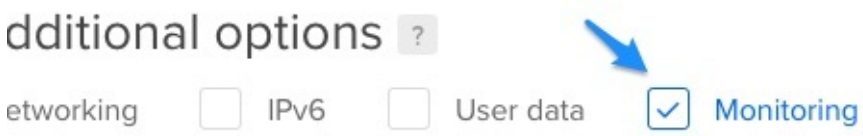


The cheapest "**1GB**" will be enough to run the site. If you have lot of **visitors** or otherwise **resource** hungry system, then you might need to *upgrade*. This package contains *free* "**monitoring**" for resources like *RAM*, *CPU* and *bandwidth*.

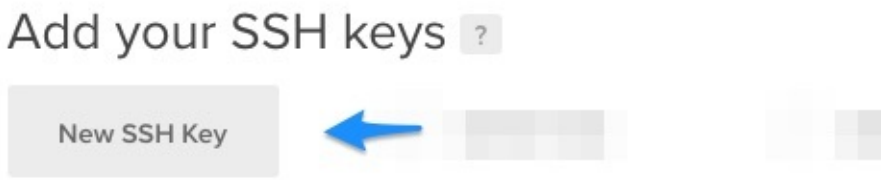


Select droplet "**location**":

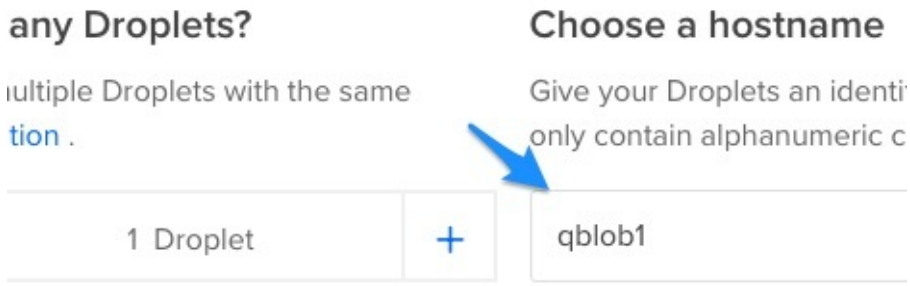
Turn on “**monitoring**”:



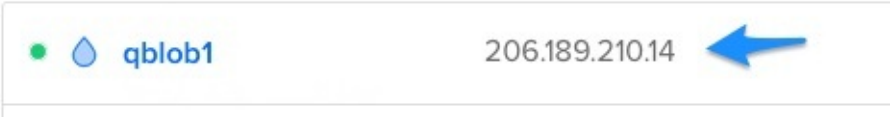
Add “**SSH**” key https://www.digitalocean.com/docs/droplets/how-to/add-ssh-keys/.



Choose a “**hostname**”:



Copy the droplet “**IP**”:



Edit your "**hosts**" file:

```
sudo vim /etc/hosts
```

In **Windows** you can find it in a location like this: "**`C:\Windows\System32\Drivers\etc\hosts`**".

Go to *insert* mode with "**i**":

Add this line (with your droplet ip):

```
206.189.210.14 qblob1
```

Exit *insert* mode with "**esc**" and save / quit with "**:wq**".

Connect to the droplet:

```
ssh root@qblob1
```

It *probably* prompts something like this:

```
The authenticity of host 'qblob1 (206.189.210.14)' can't be
established.
```

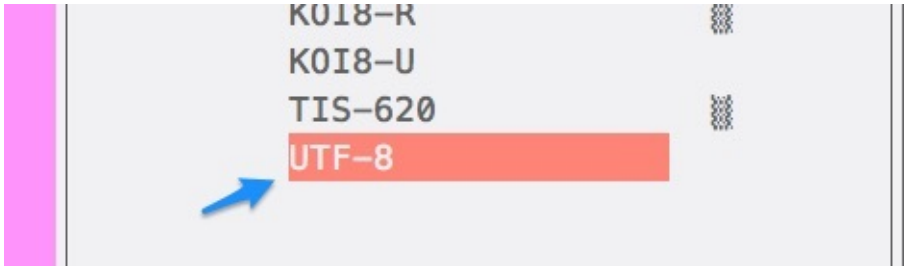
Answer "**yes**" to this question:

```
Are you sure you want to continue connecting (yes/no)?
```

Update the server:

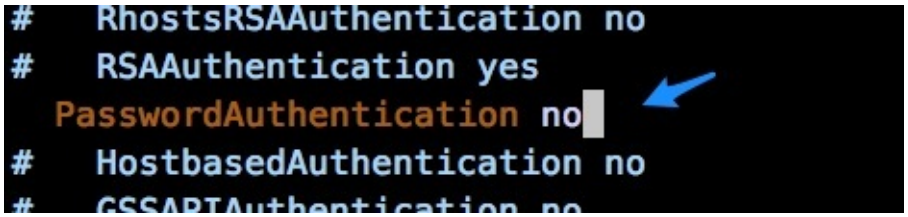
```
sudo apt-get update && sudo apt-get dist-upgrade
```

Leave “**default settings**” for all the questions like *encoding* and *menu.lst*. Use “**tab**” to jump to the **ok** button:



You can **disable login with username & password** by setting “**PasswordAuthentication**” to **no**:

```
vim /etc/ssh/ssh_config
```



Now you can “**only**” login with **ssh**.

> **Don’t worry** if you make a mistake and can’t login to the system. At this point you can just destroy and re-create the droplet.

Unix User

Let’s create a “**user**” for the site:

```
root@qblob1:~# sudo adduser qblob1
```

Setup a **good password** for the user. I’m using username and password “**qblob1**” in this example.

Leave other **user information** fields to default values.

Add user to the "**sudo**" group:

```
root@qblob1:~# sudo adduser qblob1 sudo
```

Change to the user:

```
root@qblob1:~# su - qblob1
```

Generate **ssh-key** (hit return for all questions):

```
qblob1@qblob1:~$ ssh-keygen
```

Open new **tab** in "**local**" terminal and copy your **public** ssh key:

```
cat ~/.ssh/id_rsa.pub
```

Create **authorized_keys** file in the "**server**":

```
qblob1@qblob1:~$ vim ~/.ssh/authorized_keys
```

Use **i** for *insert* mode and **paste** your public key there.

Use **esc** to exit *insert* mode and **:wq** to *save and exit*.

Leave the connection open and login using another tab:

```
ssh qblob1@qblob1
```

For now on you can login to the **production** server with "**ssh qblob1@qblob1**" instead of "*ssh root@qblob1*".

It might say "**System restart required**". Let's try that:

```
sudo reboot now
```

The *original* connection (in another tab) will the closed too:

```
Connection to qblob1 closed.
```

Freeze Requirements

Go to your "**local**" site root and "**freeze**" requirements:

```
pip freeze > requirements.txt
```

This will generate a "**requirements.txt**" file that has contents similar to this:

```
Django==2.0.7
django-appconf==1.0.2
django-imagekit==4.0.2
pilkit==2.0
Pillow==5.2.0
pytz==2018.5
selenium==3.13.0
six==1.11.0
```

We will use this information to install the same packages in the **production** server.

> At this point I'm *assuming* that you setup **GIT** according to the "**Version Control With GIT**" chapter.

Make a commit:

```
git add .
git commit -m "Add requirements.txt"
git push
```

Login to your **production** server:

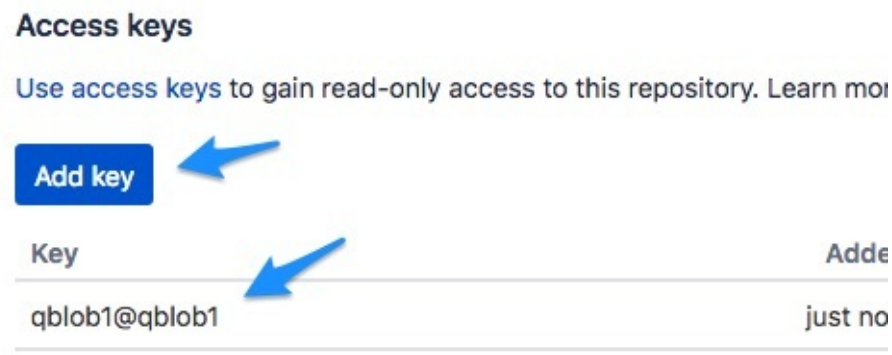
```
ssh qblob1@qblob1
```

Copy the **public ssh key**:

```
qblob1@qblob1:~$ cat ~/.ssh/id_rsa.pub
```

Go to https://bitbucket.org/YOUR-ACCOUNT/quarkblob-book/admin/access-keys/ and hit "**Add key**".

Paste the *qblob1* user public ssh key there:



Clone To Production

Now we can "**clone**" the project code to the production server.

```
qblob1@qblob1:~$ git clone git@bitbucket.org:YOUR-
ACCOUNT/quarkblob-book.git \
quarkblob
```

Project code should now be in the */home/qblob1/quarkblob/* folder.

Virtual Environment

Create the virtual environment:

```
sudo apt-get install python3-venv
export LC_ALL="en_US.UTF-8"
export LC_CTYPE="en_US.UTF-8"
```

```
python3 -m venv venv
source venv/bin/activate
```

> "**LOCALES**" | You might get some issues if you don't setup *locales* with "**export LC_ALL="en_US.UTF-8""** and "**export LC_CTYPE="en_US.UTF-8""**: https://help.ubuntu.com/community/Locale

Install project "**packages**":

```
cd quarkblob
pip install -r requirements.txt
pip install --upgrade pip
```

Settings File

We added **settings.py** file to the *.gitignore* file so it's not pulled to the production server. Let's add it manually.

Open the *local* **settings.py** in *quarkblob/project/project* and copy the contents.

Create the settings file in the *production* server and paste the contents there:

```
vim /home/qblob1/quarkblob/project/settings.py
```

Change "**DEBUG**" to *False*. Add **127.0.0.1** and **DROPLET IP** to the "**ALLOWED_HOSTS**" list:

```
DEBUG = False
ALLOWED_HOSTS = ['localhost', '127.0.0.1', '206.189.210.14']
```

> “**DEBUG MODE**” | Always set *Debug* as **False** for **live** sites! The debug mode does many things like displays detailed **error pages** that you don’t want to expose to random visitors. You can temporarily set this to “**True**” if you are getting server errors and can’t figure out the problem.

Test With Runserver & Links

You can test if the site works by running the “**development server**”:

```
python manage.py makemigrations && python manage.py migrate
python manage.py runserver
```

Open a second tab and make another connection:

```
ssh qblob1@qblob1
sudo apt-get install links
links http://127.0.0.1:8000/
```

> “**LINKS**” is a web browser that works in the *terminal*: http://links.twibright.com/user_en.html.

If everything is working you should now see something like this:

```
QuarkBlob
  * Home
  * Add post
  * About
menu
  * Home
  * About
  * login / register
login / register
Thoughts On Artificial Intelligence.
```

Use “**q**” to exit *Links*.

PostgreSQL

Install **PostgreSQL**:

```
sudo apt-get install postgresql postgresql-contrib
sudo -u postgres createuser qblob1 -P
sudo -u postgres createdb qblob1 -O qblob1
```

I used the *password* “**qblob1**” for the *qblob1* user.

> “**POSTGRESQL**” is an open source *object-relational database*: https://www.postgresql.org/. You shouldn’t use the default *SQLite* database with production sites.

If you are getting “**perl: warning: Setting locale failed.**” warning, you can try to comment “**SendEnv LANG LC_**” in “*ssh_config*”:

```
sudo vim /etc/ssh/ssh_config
# SendEnv LANG LC_*
```

Edit **settings.py**:

```
vim /home/qblob1/quarkblob/project/settings.py
```

Change the "**DATABASES**" configuration:

```
#DATABASES = {
#    'default': {
#        'ENGINE': 'django.db.backends.sqlite3',
#        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
#    }
#}

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'qblob1',
        'USER': 'qblob1',
        'PASSWORD': 'qblob1',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Install "**psycopg2**" package. Remember to activate the virtual environment first:

```
source /home/qblob1/venv/bin/activate
pip install psycopg2
cd /home/qblob1/quarkblob/
python manage.py makemigrations && python manage.py migrate
```

> "**PSYCOPG2**" is a **Python PostgreSQL adapter**: http://initd.org/psycopg/.

Restart the development server in another tab by using "**Ctrl + C**" and "**python manage.py runserver**".

Remove the “**db.sqlite3**” database and test the site with links again:

```
rm db.sqlite3
links http://127.0.0.1:8000/
```

We have now succesfully switched to using **PostgreSQL** database.

Static & Media Files

We are going to serve “**static**” files (css, js) and “**media**” files (blog images) from paths that are *not* inside the project folder.

Create the folders:

```
mkdir /home/qblob1/static
mkdir /home/qblob1/media
```

Edit **settings.py**:

```
vim /home/qblob1/quarkblob/project/settings.py
```

Specify the following **variables**:

```
STATIC_URL = '/static/'
STATIC_ROOT = '/home/qblob1/static/'
MEDIA_URL = '/media/'
MEDIA_ROOT = '/home/qblob1/media/'
```

```
cd /home/qblob1/quarkblob
python3 manage.py collectstatic
```

You should get a notice like this: “**141 static files copied to ‘/home/qblob1/static’**”.

Nginx

> We will use "**Nginx**" (https://www.nginx.com/) as a *reverse proxy* to "**Gunicorn**" *server* (http://gunicorn.org/).

```
sudo apt-get update && sudo apt-get install nginx
sudo ufw app list
sudo ufw allow 'Nginx HTTP'
sudo ufw allow 'Nginx HTTPS'
sudo ufw allow ssh
sudo ufw enable
sudo ufw status
systemctl status nginx
```

> "**UFW**" is the default **firewall** configuration tool for Ubuntu.

> We will use "**SYSTEMCTL**" to inspect and manage services: https://www.freedesktop.org/software/systemd/man/systemctl.html.

Visit the server IP **https://206.189.210.14** and you should see the **nginx** welcome screen:

Here are some useful commands for *nginx*:

```
sudo systemctl stop nginx
sudo systemctl start nginx
sudo systemctl restart nginx
# Reload Nginx configuration without restart:
sudo systemctl reload nginx
```

Setup Gunicorn

> "**GUNICORN**" is a Python **WSGI HTTP Server** for UNIX. I'm going to show you an example how to get started. Check the official documentation for details http://docs.gunicorn.org/en/latest/install.html.

Let's **install** gunicorn and create a folder where to put all related files:

```
pip install gunicorn
mkdir /home/qblob1/gunicorn
```

We could run gunicorn with a console command but it's easier *manage* a "**script**" with all the settings and commands.

Create the script:

```
vim /home/qblob1/gunicorn/gunicorn_launch
```

Add these lines inside:

```
#!/bin/bash
NAME="qblob1"
BIND=unix:/home/qblob1/gunicorn/gunicorn.sock
WORKERS=3

USER=qblob1
GROUP=qblob1
```

```
DIR=/home/qblob1/quarkblob
DJANGO_SETTINGS_MODULE=project.settings
DJANGO_WSGI_MODULE=project.wsgi

LOG_LEVEL=error
LOG_FILE=/home/qblob1/logs/gunicorn-error.log

cd $DIR
source ../venv/bin/activate

export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DIR:$PYTHONPATH

exec ../venv/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name $NAME \
  --workers $WORKERS \
  --user=$USER \
  --group=$GROUP \
  --bind=$BIND \
  --log-level=$LOG_LEVEL \
  --log-file=$LOG_FILE
```

Make the script an **executable**:

```
chmod u+x /home/qblob1/gunicorn/gunicorn_launch
```

Create **error** log files:

```
mkdir /home/qblob1/logs/
touch /home/qblob1/logs/gunicorn-error.log
touch /home/qblob1/logs/nginx-access.log
touch /home/qblob1/logs/nginx-error.log
```

Check these files out if you run into problems later!

Edit **Nginx** configuration file:

```
sudo vim /etc/nginx/sites-available/default
```

Use “**gg dG**” in Vim to *delete everything*, goto *insert* mode with “**i**” and paste this in it (use your droplet ip as the *server_name*):

```
upstream app_server {
    server unix:/home/qblob1/gunicorn/gunicorn.sock
fail_timeout=0;
}

server {
    listen 80;
    server_name 206.189.210.14;

    keepalive_timeout 5;
    client_max_body_size 4G;
    access_log /home/qblob1/logs/nginx-access.log;
    error_log /home/qblob1/logs/nginx-error.log;

    location /static/ {
        alias /home/qblob1/static/;
    }

    location /media/ {
        alias /home/qblob1/media/;
    }

    location / {
        try_files $uri @proxy_to_app;
    }

    location @proxy_to_app {
      proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_redirect off;
      proxy_pass http://app_server;
    }
}
```

Checkout more options here:
http://docs.gunicorn.org/en/latest/deploy.html#nginx-configuration

Reload Nginx configuration:

```
sudo systemctl reload nginx
```

> Sometimes you might need to **restart** nginx like this: "**sudo systemctl restart nginx**".

Run **gunicorn**:

```
cd /home/qblob1/gunicorn
./gunicorn_launch
```

Visit the droplet ip and it should be serving you the website with *Nginx* and *Gunicorn*.

> "**DATABASE VALUES**" | We will use **fixtures** in the next chapter to load the **editor** group and **tags** to the database. Don't create any content yet to the db, not even the *superuser*.

Monitor Gunicorn

We want all services to start up automatically on reboot. **Nginx** and **PostgreSQL** does this by default. Let's run the "**gunicorn_launch**" script on boot as well.

> There are plenty of **process monitoring systems**. I'm going to use "**SUPERVISOR**": http://supervisord.org/.

```
sudo apt-get -y install supervisor
sudo systemctl enable supervisor
sudo systemctl start supervisor
sudo systemctl status supervisor
```

Create the project supervisor **configuration** file. It will be included automatically to the supervisor main configuration file if you put it here:

```
sudo vim /etc/supervisor/conf.d/qblob1.conf
```

Put these lines in it:

```
[program:qblob1]
command=/home/qblob1/gunicorn/gunicorn_launch
user=qblob1
autostart=true
autorestart=true
redirect_stderr=true
stdout_logfile=/home/qblob1/logs/gunicorn-error.log
```

Reload:

```
sudo supervisorctl reload
sudo supervisorctl update
sudo supervisorctl status qblob1
```

Reboot the machine:

```
sudo reboot now
```

Wait a while and visit the droplet **IP** after the reboot is done and the website should be still working.

Use this command when you need to **restart** gunicorn:

```
sudo supervisorctl reload qblob1
```

> **RESTART GUNICORN PROCESS** when you make **changes** to the site or you could be serving an *old* build.

Password Protection With .htpasswd

You can setup a password with "**.htpasswd**" to *restrict access* to the site:

```
sudo sh -c "echo -n 'qblob1:' >> /etc/nginx/.htpasswd"
sudo sh -c "openssl passwd -apr1 >> /etc/nginx/.htpasswd"
```

Edit **Nginx** configuration file:

```
sudo vim /etc/nginx/sites-available/default
```

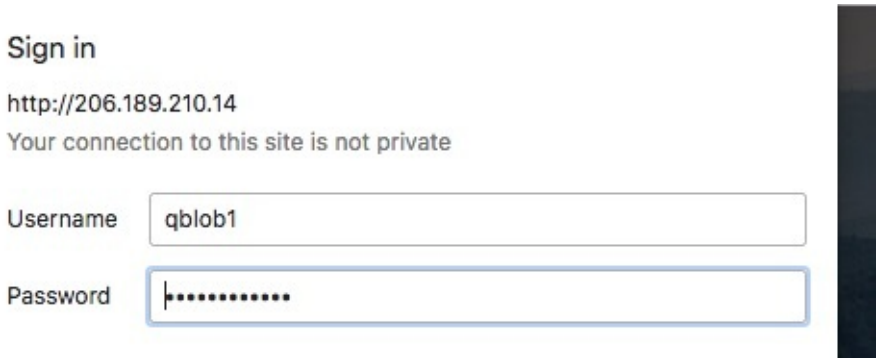
Add following "**auth_**" lines:

```
location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a
404.
                try_files $uri $uri/ =404;
                auth_basic "Restricted Content";
                auth_basic_user_file /etc/nginx/.htpasswd;
        }
```

Reload the configuration:

```
sudo systemctl reload nginx
```

Now you will be asked for **username** and **password** when visiting the ip:



Deploy Changes

Let's test the process of **making changes** and **deploying** those changes to the live server.

Edit "**base.html**" in *quarkblob/project/base/templates/base/* and add some text inside the main container:

```
<div class="main {% block hidebox %}{% endblock %}">
    <h1 style="font-size: 50px;">TEST</h1>
    {% block h1 %}{% endblock %}
    {% block content %}{% endblock %}
</div>
```

Push the changes:

```
git add .
git commit -m "Add test text in the main container"
git push
```

Login to the live site and **pull** the changes:

```
ssh qblob1@qblob1
cd /home/qblob1/quarkblob
git pull
```

You might also need to **collect static files** (for example if you do *CSS* changes) and restart **gunicorn** for all changes to show up so it's useful to run the following line of commands at once.

```
cd /home/qblob1/quarkblob && \
git pull &&  \
source /home/qblob1/venv/bin/activate && \
python manage.py collectstatic --noinput && \
sudo supervisorctl reload qblob1
```

> “**MERGING**” | If an editor like **Nano** opens and prompts you a merge message, just save the file with **Ctrl + O**. If you have *merge conflicts* then you have to **resolve** those first: https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/

> “**CTRL + R**” allows you to **search previous commands** in Linux. This will save you a lot of time.

Let’s “**reverse**” the change:

Edit “**base.html**” in *quarkblob/project/base/templates/base/* and remove the “**TEST**” text:

```
<div class="main {% block hidebox %}{% endblock %}">
    <h1 style="font-size: 50px;">TEST</h1>
    {% block h1 %}{% endblock %}
    {% block content %}{% endblock %}
</div>
```

Push changes:

```
git add .
git commit -m "Remove test text"
git push
```

Pull and reload…

```
cd /home/qblob1/quarkblob && \
git pull &&  \
source /home/qblob1/venv/bin/activate && \
python manage.py collectstatic --noinput && \
sudo supervisorctl reload qblob1
```

…and the “TEST” text will be gone.

> “**BROWSER CACHE**” | Remember to keep *developer tools* open in **Chrome** so you don’t load the pages from the browser cache.

> “**CI**” | We did the whole “**deployment process**” *manually* but take a look at “**Continuous Integration and Automation**” systems like *Jenkins* and *Ansible*.

Settings For Production

We added “**settings.py**” manually to the production site but you might want to create a “**local_settings.py**” file that contains server *specific* configuration. Next I will show you *one* way to do it.

Edit *local* “**settings.py**” in *quarkblob/project/project/* and add these lines at the bottom:

```
try:
    from .local_settings import *
except ImportError:
    pass
```

Edit “**.gitignore**” in *quarkblob/project/* and do these changes:

```
# DJANGO
__pycache__/
*.py[cod]
db.sqlite3
settings.py
local_settings.py
# OTHER
node_modules
```

```
.DS_Store
.idea
media
```

Push the settings file to the repo:

```
git add .
git commit -m "Update settings.py"
git push
```

> "**COPY THE SETTINGS.PY DATA**" | Copy the production site "**settings.py**" file data at this point because next we will override it.

Login to the "**production**" site and pull the settings file:

```
ssh qblob1@qblob1
cd quarkblob
git pull
```

Create "**local_settings.py**" in */home/qblob1/quarkblob/project/* and add "**production**" server specific data there:

```
DEBUG = False
ALLOWED_HOSTS = ['localhost', '127.0.0.1', '142.93.80.146',
'www.quarkblob.co\
m']

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'qblob1',
        'USER': 'qblob1',
        'PASSWORD': 'qblob1',
        'HOST': 'localhost',
        'PORT': '',
    }
}

STATIC_URL = '/static/'
```

```
STATIC_ROOT = '/home/qblob1/static/'
MEDIA_URL = '/media/'
MEDIA_ROOT = '/home/qblob1/media/'
```

The website should work as before but it overrides some of the **settings.py** file configuration by importing the "**local_settings.py**" file contents.

You might need to *restart* the processes:

```
sudo supervisorctl restart qblob1
```

25. Initial Data With Fixtures

> With "**FIXTURES**" you can *serialize* contents of the database. **Serialization** translates object data to a format that is easier to transfer and *reconstruct* later. In this case we store the object data in *JSON* format.

Dump Data

I'm assuming that you haven't created any data in the production database yet (not even superuser).

Go to your "**local**" site *root* and **dump** the data:

```
python manage.py dumpdata auth.User auth.Group blog.Tag --
indent 4 --natural-\
foreign --natural-primary -o data.json
```

We are only dumping "**Users**", "**Groups**" and "**Tags**".

"**–indent**" will format the file in more **readable** form.

We need to specify the "**natural**" options otherwise the *Editor* group **permissions** might not be correct.

Read more about *natural keys*: https://docs.djangoproject.com/en/dev/topics/serialization/#topics-serialization-natural-keys.

"**-o data.json**" specifies the **output** file.

> I'm dumping the **user** data for *convenience* but bare in mind that the password *hashes* will be also stored in the file. You might want to leave "**auth.User**" out and create the production site superuser yourself "**python manage.py createsuperuser**".

Read more about *dumpdata*: https://docs.djangoproject.com/en/dev/ref/django-admin/#dumpdata.

"**Push**" the data:

```
git add .
git commit -m "Add initial data"
git push
```

Load Data

"**Pull**" and use "**loaddata**" to load the data into the database:

```
ssh qblob1@qblob1
source /home/qblob1/venv/bin/activate
cd quarkblob
git pull
python manage.py loaddata data.json
```

Read more about *loaddata*: https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-loaddata.

You should see something like this:

```
Installed 6 object(s) from 1 fixture(s)
```

Now you can start creating content without needing to setup the "**Editor**" group, "**users**" or "**tags**". Any *testusers* and their

permissions are now transferred to the *production* site.

Read more about *fixtures*: https://docs.djangoproject.com/en/dev/howto/initial-data/#providing-data-with-fixtures.

> You can also use "**MIGRATIONS**" to change the database data: https://docs.djangoproject.com/en/dev/topics/migrations/#data-migrations.

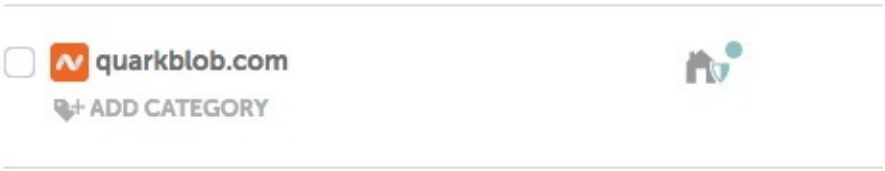
26. Domain

Eventually you want to buy a “**domain**” name so people don’t have to use the **ip** address to access your site.

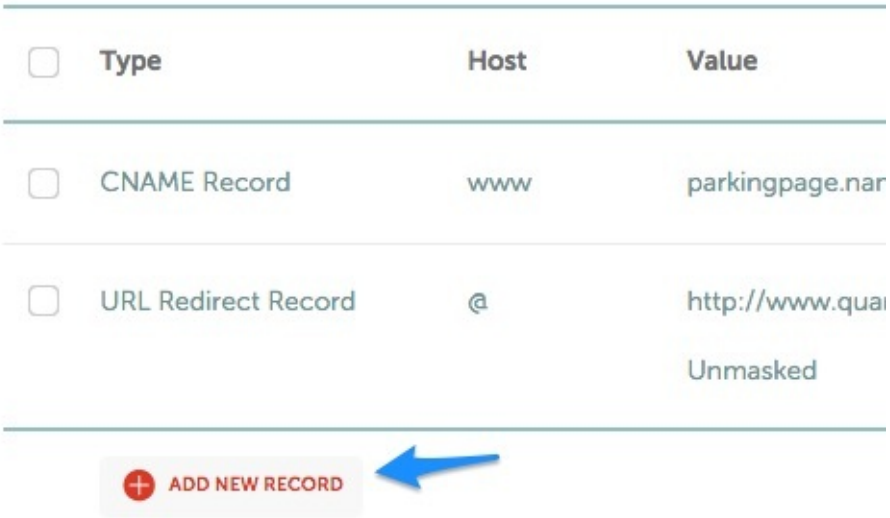
Namecheap

I have used https://www.namecheap.com/ for lot of my international domains. Let’s configure a domain with it. The configuration is pretty much the same *regardless* of the domain provider.

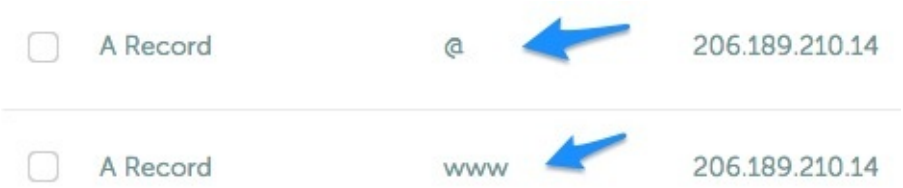
Buy a domain:



Manage the domain and select “**Advanced DNS**” settings. Select “**ADD NEW RECORD**”.



Add “**A**” *records* for “**@**” and “**www**” (use the droplet ip):



You can remove the **CNAME** *record* for “**parkingpage.namecheap.com**”.

Nginx

Login to the server:

```
ssh qblob1@qblob1
```

Edit **Nginx** configuration:

```
sudo vim /etc/nginx/sites-enabled/default
```

Hit “**i**” to get into insert mode and a **block** to redirect **non-www** request to **www** request. Also change the “**server_name**” for the current server block:

```
upstream app_server {
    server unix:/home/qblob1/run/gunicorn.sock fail_timeout=0;
}

server {  # redirect here
    listen 80;
    server_name quarkblob.com;
    return 301 $scheme://www.quarkblob.com$request_uri;
}

server {
    listen 80;
    server_name www.quarkblob.com;  # replace the ip with your
domain name
```

Save and quit with “**esc**” and “**:wq**”.

Reload Nginx:

```
sudo service nginx restart
```

Edit **settings.py** (or *local_settings.py* if you use that):

```
vim /home/qblob1/quarkblob/project/settings.py
```

Add domain in “**ALLOWED_HOSTS**”:

```
ALLOWED_HOSTS = ['localhost', '127.0.0.1', '206.189.210.14',
'www.quarkblob.c\
om']
```

Restart *supervisor*:

```
sudo supervisorctl reload
```

You should now be able to access the website with the domain name.

> “**DOMAIN PROPAGATION**” | Changes to DNS records do not *propagate* throughout the network immediately so you might have to **wait** for some time until it starts associating the domain name with the droplet ip.

27. HTTPS And Nginx

It's a good idea to **encrypt** the communication between the browser and your website. Even if you are not dealing with any sensitive data, this adds *trustworthiness* to the service.

Let's Encrypt

> "**Let's Encrypt**" is a **certificate authority** that provides *free* certificates to secure websites.

```
sudo apt-get update
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get update
sudo apt-get install python-certbot-nginx
```

```
sudo certbot --nginx
```

Enter "**email address**" (used for urgent renewal and security notices).

Select the domain with "www" (**option 2**) when it asks something like this:

```
Which names would you like to activate HTTPS for?
1: quarkblob.com
2: www.quarkblob.com
```

Make it "**redirect**" requests to HTTPS by selecting "redirect" (**option 2**):

```
2: Redirect - Make all requests redirect to secure HTTPS
access.
```

You can test the **automatic renewal** for your certificates by running this command:

```
sudo certbot renew --dry-run
```

Firewall

Make sure you have **Nginx HTTPS** enabled for the **firewall**:

```
sudo ufw status
sudo ufw allow 'Nginx HTTPS'
```

Refresh the page and you should see the "**green padlock**":

Check the official website for more info: https://certbot.eff.org

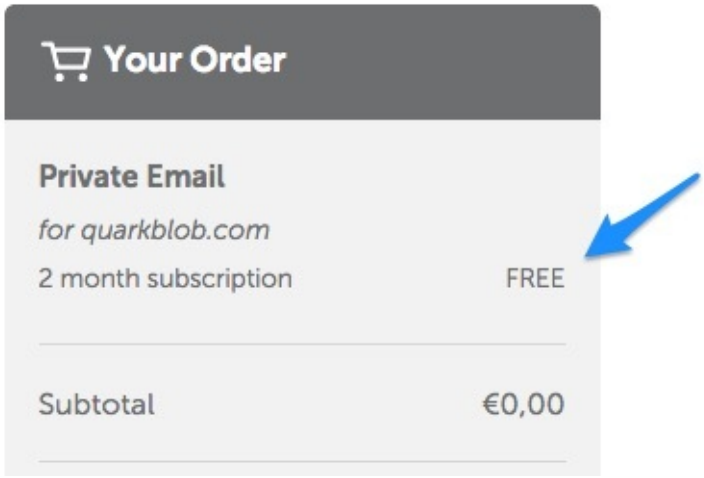
28. Send Email

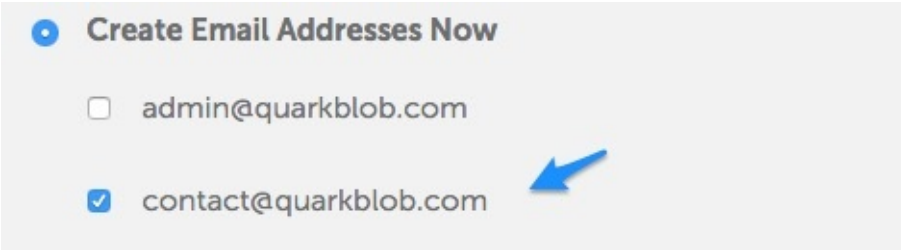
Setup Mailbox

You need to setup a “**mailbox**” if you want to send **password reset** links or other emails.

You can try **Namecheap** email service for free. Go to https://www.namecheap.com/hosting/email.aspx and select “**Private**” solution for “**2 months**”.

Select “**Use a domain I own with Namecheap**” if you have bought a domain with Namecheap.





- Go to “**domains**” list

- “**Manage**” domain
- Go to “**Products**”
- Manage “**Mailbox**”
- “**Set Password**” for the mailbox

Edit the *production* server “**settings.py**” (or *local_settings.py*) in *quarkblob/project/project/* and add “**EMAIL**” configuration:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'mail.privateemail.com'
EMAIL_HOST_USER = 'contact@quarkblob.com'
EMAIL_HOST_PASSWORD = '******'
EMAIL_PORT = '465'
EMAIL_USE_SSL = True
DEFAULT_FROM_EMAIL = 'QuarkBlob.com <contact@quarkblob.com>'
SERVER_EMAIL = 'contact@quarkblob.com'
```

Create “**superuser**” for the *production* site (if you haven’t already created it):

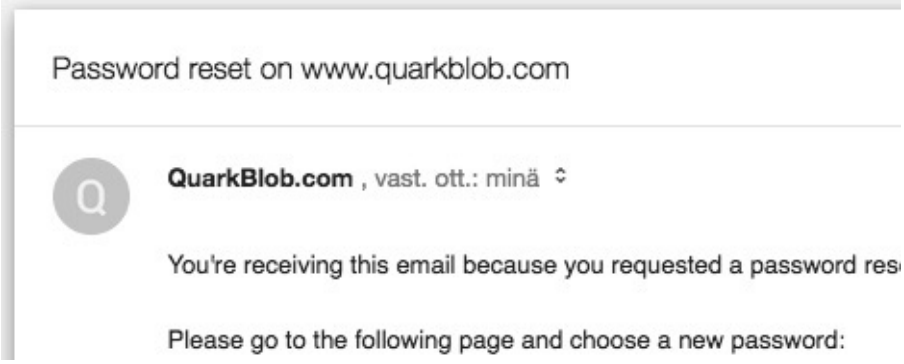
```
python manage.py createsuperuser
```

Restart the **qblob1** *process group*:

```
sudo supervisorctl restart qblob1
```

Visit “**accounts/password_reset/**” and fill in the admin user email.

You should now receive a “**reset link**” to your admin user email address:



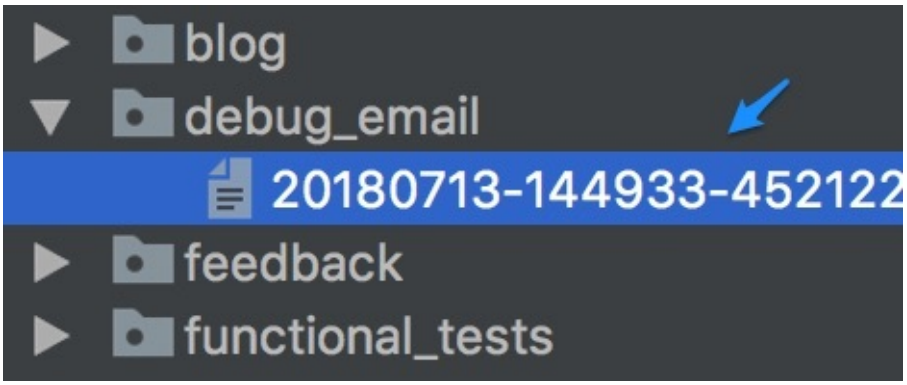
Email Logging In Localhost

You can use the same email settings in *development* to send mail but it's usually more useful to "**log**" the emails to a file:

Edit "**settings.py**" in *quarkblob/project/project/* and set "**EMAIL_BACKEND**" and "**EMAIL_FILE_PATH**" variables:

```
EMAIL_BACKEND =
"django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = os.path.join(BASE_DIR, "debug_email")
```

Now try to **reset** password in *localhost* and the mail will be stored in the "**debug_email**" folder:



29. Caching

As the website size grows, it might start to use considerable amount of resources. One easy way to increase **performance** is to **cache** things. **Dynamic** systems like Django might calculate certain things over and over again when visitors request pages. Some of this can be *cached* so the requested information is loaded straight from the cache, bypassing these calculations.

Installation

> "**MEMCACHED**" is an open source *memory object caching system:* https://memcached.org/. This is the **fastest** cache Django supports *natively* and pretty *easy* to setup.

Install "**memcached**" and "**python-memcached**" package in the *production* server:

```
sudo apt-get install memcached
source /home/qblob1/venv/bin/activate
pip install python-memcached
```

Configuration

Edit project "**settings.py**" (or *local_settings.py*) file:

```
vim /home/qblob1/quarkblob/project/settings.py
```

Define “**CACHES**” configuration and add “**‘django.middleware.cache.UpdateCacheMiddleware’,**” on **top** of the *MIDDLEWARE* list and “**‘django.middleware.cache.FetchFromCacheMiddleware’,**” at the **bottom** of the list:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}

MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    ...
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

Reboot and you should see memcached still running:

```
sudo reboot now
ssh qblob1@qblob1
sudo netstat -tupan | grep 11211
```

Visit some pages on the site and use these commands to see if the caching is really working:

```
/usr/share/memcached/scripts/memcached-tool 127.0.0.1:11211
display
/usr/share/memcached/scripts/memcached-tool 127.0.0.1:11211
stats
```

For example with the second command you can see “**curr_items**” changing as you click new pages to be cached.

> “**NOTE**” | This was a very *brief* introduction to caching. You might want to be more **granular** with it and not cache the **whole website**: https://docs.djangoproject.com/en/dev/topics/cache/#the-per-view-cache.

30. Afterword

Congratulations!

The purpose of this book was to introduce you to a **practical process** that you can start applying right **now** to create awesome apps. Of course this is just the beginning. Many Django topics weren't covered *in depth* and other technologies were only scratched.

Where To Go Next?

Web development is an *exciting* and *dynamic* pursuit. The next step depends on where you want to go. There are tons of interesting areas you can lost yourself for years. Learn **JavaScript** if you are interested in the *frontend* of things. If you want to *really* understand the backend, focus on **Python** and look deeper into the **Django** framework. Increasing your **Programming** skills and **Computer Science** knowledge in general is a good investment. Look into **Server** / **Database** technologies and **Continuous Integration** systems. If you are starting a **Business** in the **Software Industry** then there are numerous other areas of knowledge outside *practical software skills* that you have to familiarize yourself with.

I hope you acquired some *insights* and *inspiration* for the journey ahead.

Checkout my tutorial website for more content: https://www.wdtutorials.com/django/.

Never stop learning.