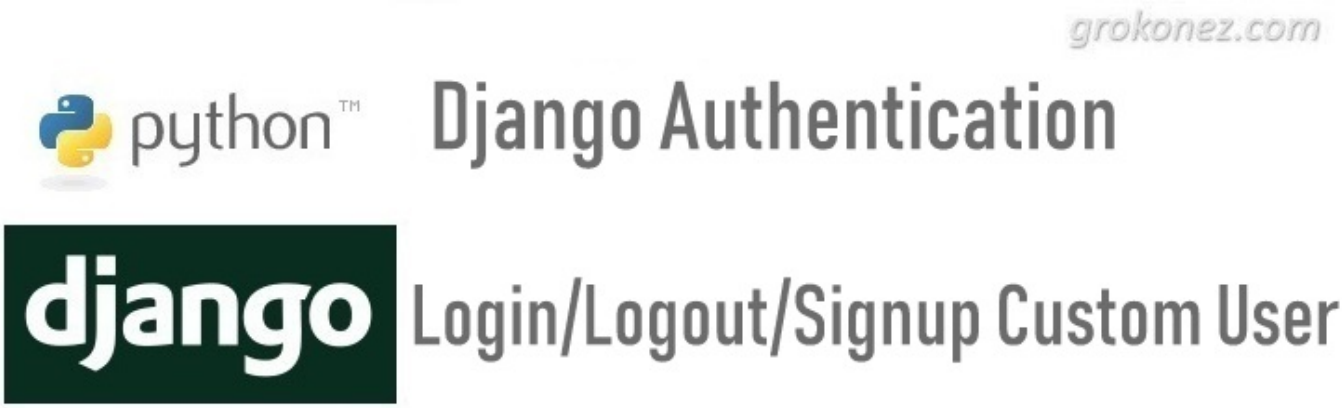
grokonez START HERE

Django Authentication – How to build Login/Logout /Signup for custom User



Building user authentication is not easy, in almost case, it's complicated. Fortunately, Django has a powerful built-in User authentication that helps us create our Authentication system fast. By default, the User model in Django `auth` app contains fields: *username*, *password*, *email*, *first_name*, *last_name*... However, using our own custom user model allows us deal with user profile more comfortably. For example, what if we want to add more fields: *full_name* or *age*?

In this tutorial, we're gonna look at way to customize authentication in Django (version 2.1) using subclass of `AbstractBaseUser`: `AbstractUser`. All User authentication data will be stored in **MySQL**/**PostgreSQL** database that we'll show you how to config the datasource.

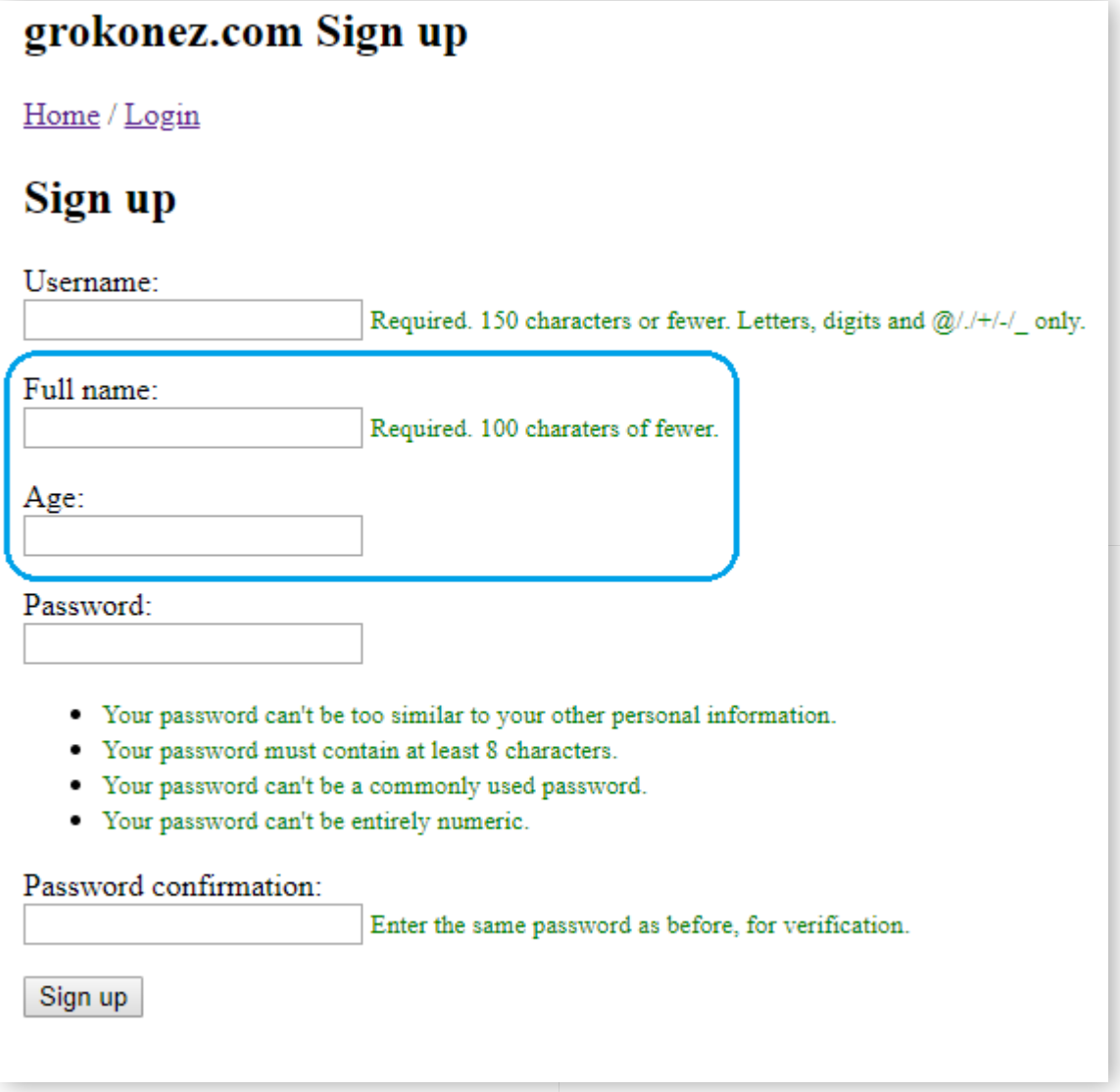
It will be very interesting. Let's go through the steps.

Contents [hide]

Django Custom Authentication Project overview

Goal

We will build a Dajngo Project with Authentication app that has login/logout /signup with custom fields such as full name and age:

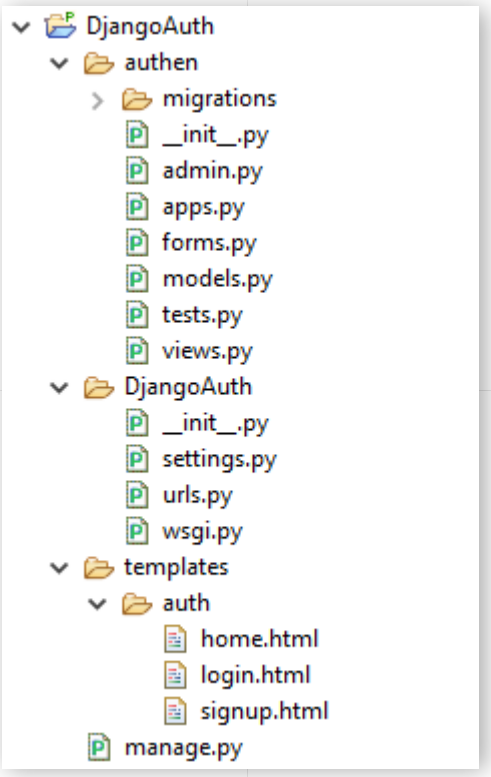


We will code our custom `signup()` function, `login()` and `logout()` is automatically implemented by Django `auth`.

All User data will be saved in MySQL/PostgreSQL database.

Project Structure

Here is the folders and files structure that we will create in the next steps.



Setup Django Custom Authentication Project

Create Django project named **DjangoAuth** with command:
`django-admin startproject DjangoAuth`

Run following commands to create new Django App named **authen** inside the project:
– `cd DjangoAuth`
– `python manage.py startapp authen`

Open ***upload/apps.py***, we can see `AuthenConfig` class (subclass of the `django.apps.AppConfig`) that represents our Django app and its configuration:

```
from django.apps import AppConfig


class AuthenConfig(AppConfig):
    name = 'authen'
```

Open ***settings.py***, find `INSTALLED_APPS`, then add:

```
INSTALLED_APPS = [
    ...
    'authen.apps.AuthenConfig',
]
```

Config Django project to work with database

MySQL Database

Install & Import Python MySQL Client

We have to install Python MySQL Client to work with MySQL database.
In this tutorial, we use **pymysql**: `pip install pymysql`.

Once the installation is successful, import this module in ***DjangoAuth/__init__.py***:

```
import pymysql

pymysql.install_as_MySQLdb()
```

Setup MySQL Database engine

Open ***settings.py*** and change declaration of `DATABASES`:

```
DATABASES = {
   'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'testdb',
        'USER': 'root',
        'PASSWORD': '123456',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

PostgreSQL Database

Install Python PostgreSQL adapter

We have to install Python PostgreSQL adapter to work with PostgreSQL database.
In this tutorial, we use **psycopg2**: `pip install psycopg2`.

Setup PostgreSQL Database engine

Open ***settings.py*** and change declaration of `DATABASES`:

```
DATABASES = {
   'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'testdb',
        'USER': 'postgres',
        'PASSWORD': '123',
        'HOST': '127.0.0.1',
```

```
        'PORT': '5432',
    }
}
```

Create Custom User Model

Create a new Custom User Model

In *authen/models.py*, create a new User model called `CustomUser` that extends `AbstractUser` (a subclass of AbstractBaseUser), then add two custom fields: `full_name` and `age`:

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


class CustomUser(AbstractUser):
    full_name = models.CharField(max_length=100, blank=False)
    age = models.PositiveIntegerField(null=True, blank=True)
```

`age` field uses both `null` and `blank`:
- `null` is for database. `null=True` indicates that we can store it in database entry as **NULL** (no value).
- `blank` is for validation. `blank=True` accepts empty value for the form field, so `blank=False` indicates that the value is required.

Specify Custom User Model in setting.py

In *setting.py*, we add `AUTH_USER_MODEL` config to specify our custom user model instead of Django built-in `User` model. The model is named `CustomUser` and exists within `authen` app, so we refer to it as `authen.CustomUser`:

```
AUTH_USER_MODEL = 'authen.CustomUser'
```

Create a new form for UserCreationForm

Now we create a new file in the authen app called *forms.py*:

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

```
from .models import CustomUser


class SignUpForm(UserCreationForm):
    full_name = forms.CharField(max_length=100, help_text='Required. 100 charaters

    class Meta:
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ('full_name', 'age',)
```

Our `SignUpForm` extends the `UserCreationForm`.

We set `model` to `CustomUser` and use default fields by `Meta.fields` which includes all default fields (including username, first_name, last_name, email, password, groups...). We simply plus our custom fields (`full_name`, `age`) at the end and it will display automatically on signup page.

When a user signs up for a new account, the default form only asks for a `username`, `email`, and `password`. Now, it also requires `full_name` and `age`.

Activate the User Model

Now our new database model is created, we need to update Django in 2 steps:

Create migration file

Run the command:
`python manage.py makemigrations authen`

We can see output text:

```
Migrations for 'authen':
  authen\migrations\0001_initial.py
    - Create model CustomUser
```

It indicates that the authen/migrations/0001_initial.py file includes code to create CustomUser data model:

```
# Generated by Django 2.1.7 on 2019-03-23 10:10

import django.contrib.auth.models
import django.contrib.auth.validators
from django.db import migrations, models
import django.utils.timezone


class Migration(migrations.Migration):
```

```
    initial = True

    dependencies = [
        ('auth', '0009_alter_user_last_name_max_length'),
    ]

    operations = [
        migrations.CreateModel(
            name='CustomUser',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serio
                ('password', models.CharField(max_length=128, verbose_name='passwo
                ('last_login', models.DateTimeField(blank=True, null=True, verbose
                ('is_superuser', models.BooleanField(default=False, help_text='Des
                ('username', models.CharField(error_messages={'unique': 'A user wi
                ('first_name', models.CharField(blank=True, max_length=30, verbose
                ('last_name', models.CharField(blank=True, max_length=150, verbose
                ('email', models.EmailField(blank=True, max_length=254, verbose_no
                ('is_staff', models.BooleanField(default=False, help_text='Designo
                ('is_active', models.BooleanField(default=True, help_text='Designo
                ('date_joined', models.DateTimeField(default=django.utils.timezone
                ('full_name', models.CharField(max_length=100)),
                ('age', models.PositiveIntegerField(blank=True, null=True)),
                ('groups', models.ManyToManyField(blank=True, help_text='The group
                ('user_permissions', models.ManyToManyField(blank=True, help_text=
            ],
            options={
                'verbose_name': 'user',
                'verbose_name_plural': 'users',
                'abstract': False,
            },
            managers=[
                ('objects', django.contrib.auth.models.UserManager()),
            ],
        ),
    ]
```

The generated code defines a subclass of the `django.db.migrations.Migration`. It has an operation for creating `CustomUser` model table. Call to `migrations.CreateModel()` method will create a table that allows the underlying database to persist the model.

You can see that we have not only user default fields but also custom fields (`full_name`, `age`).

Generate database table

Run the following Python script to apply the generated migration:

```
python manage.py migrate
```

The output text:

```
Operations to perform:
  Apply all migrations: admin, auth, authen, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying authen.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying sessions.0001_initial... OK
```

Check MySQL Database, for example, now we can see that a table for CustomUser model was generated and it's named `authen_customuser`:

```
mysql> describe authen_customuser;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| id           | int(11)          | NO   | PRI | NULL    | auto_increment |
| password     | varchar(128)     | NO   |     | NULL    |                |
| last_login   | datetime(6)      | YES  |     | NULL    |                |
| is_superuser | tinyint(1)       | NO   |     | NULL    |                |
| username     | varchar(150)     | NO   | UNI | NULL    |                |
| first_name   | varchar(30)      | NO   |     | NULL    |                |
| last_name    | varchar(150)     | NO   |     | NULL    |                |
| email        | varchar(254)     | NO   |     | NULL    |                |
| is_staff     | tinyint(1)       | NO   |     | NULL    |                |
| is_active    | tinyint(1)       | NO   |     | NULL    |                |
| date_joined  | datetime(6)      | NO   |     | NULL    |                |
| full_name    | varchar(100)     | NO   |     | NULL    |                |
| age          | int(10) unsigned | YES  |     | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
```

Set urlpatterns & handle signup/login/logout requests

Set url patterns

Open the project-level *urls.py* file, we're gonna use built-in `django.contrib.auth` module to handle login/logout requests:

```
from django.urls import path
from django.contrib.auth import views as auth_views
```

```
from authen import views

urlpatterns = [
    path('', views.home, name='home'),
    path('login/', auth_views.LoginView.as_view(template_name='auth/login.html'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('signup/', views.signup, name='signup'),
]
```

When passing `'auth/login.html'` for `template_name` parameter, we tell auth.views.LoginView that the path will be used for login form (we will create later in this tutorial).

Using `auth.views.LogoutView` helps us handle logout request automatically, we only need to call `{% url 'logout' %}` where we want to make logout event in the HTML template.

The next step is to specify where to redirect the user upon a successful login/logout.
Open project *setting.py*, then set values for `LOGIN_REDIRECT_URL` and `LOGOUT_REDIRECT_URL`:

```
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home'
```

Now, after login/logout, if we don't indicate where to come, the user will be redirected to the `'home'` template which is our homepage.

Custom signup request

Inside *authen/views.py*, define functions for handling signup request and homepage:

```
from django.shortcuts import render, redirect
from django.contrib.auth import login, authenticate

from .forms import SignUpForm


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


def signup(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            user = form.save()
```

```
            user.save()
            raw_password = form.cleaned_data.get('password1')
            user = authenticate(username=user.username, password=raw_password)
            login(request, user)
            return redirect('home')
    else:
        form = SignUpForm()
    return render(request, 'auth/signup.html', { 'form' : form })
```

Now we dive into `signup()` function. It gets user data from HTTP **POST** request which is handled by `SignUpForm`, save user to database.

Then we use `authenticate()` function and `login()` function from `django.contrib.auth` to log the user in.

If the process is successful, redirect to homepage, otherwise, return to `signup.html` template.

Create Django template for User Authentication

Specify template directory

In project's *settings.py*, set templates path for `'DIRS'`:

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

Create Homepage template

We're gonna use Django built-in [url templatetag]() to create links for login/logout /signup requests.

In project-level folder, create new **templates** folder, then create new HTML file named *home.html*:

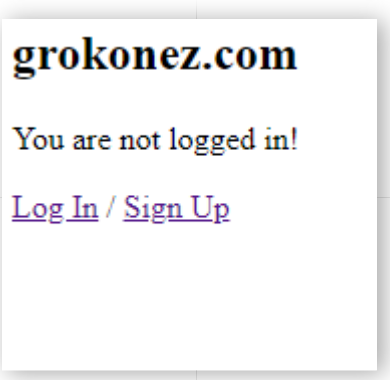
```
<h2>grokonez.com</h2>

{% if user.is_authenticated %}
  <p>Hi {{ user.full_name }}, Welcome to grokonez!</p>
  <p><a href="{% url 'logout' %}">Log Out</a></p>
{% else %}
```

```
  <p>You are not logged in!</p>
  <a href="{% url 'login' %}">Log In</a> /
  <a href="{% url 'signup' %}">Sign Up</a>
{% endif %}
```

We can use `is_authenticated` attribute to specify whether the user is logged in or not, then show his full name for a website greeting.

So, when user aren't logged in, it looks like:



Create Django custom Signup template

– We're gonna use HTML `<form>` tag with HTTP POST method.
– We add `{% csrf_token %}` to protect our form from cross-site scripting attacks.

In **templates** folder, create *signup.html* file:

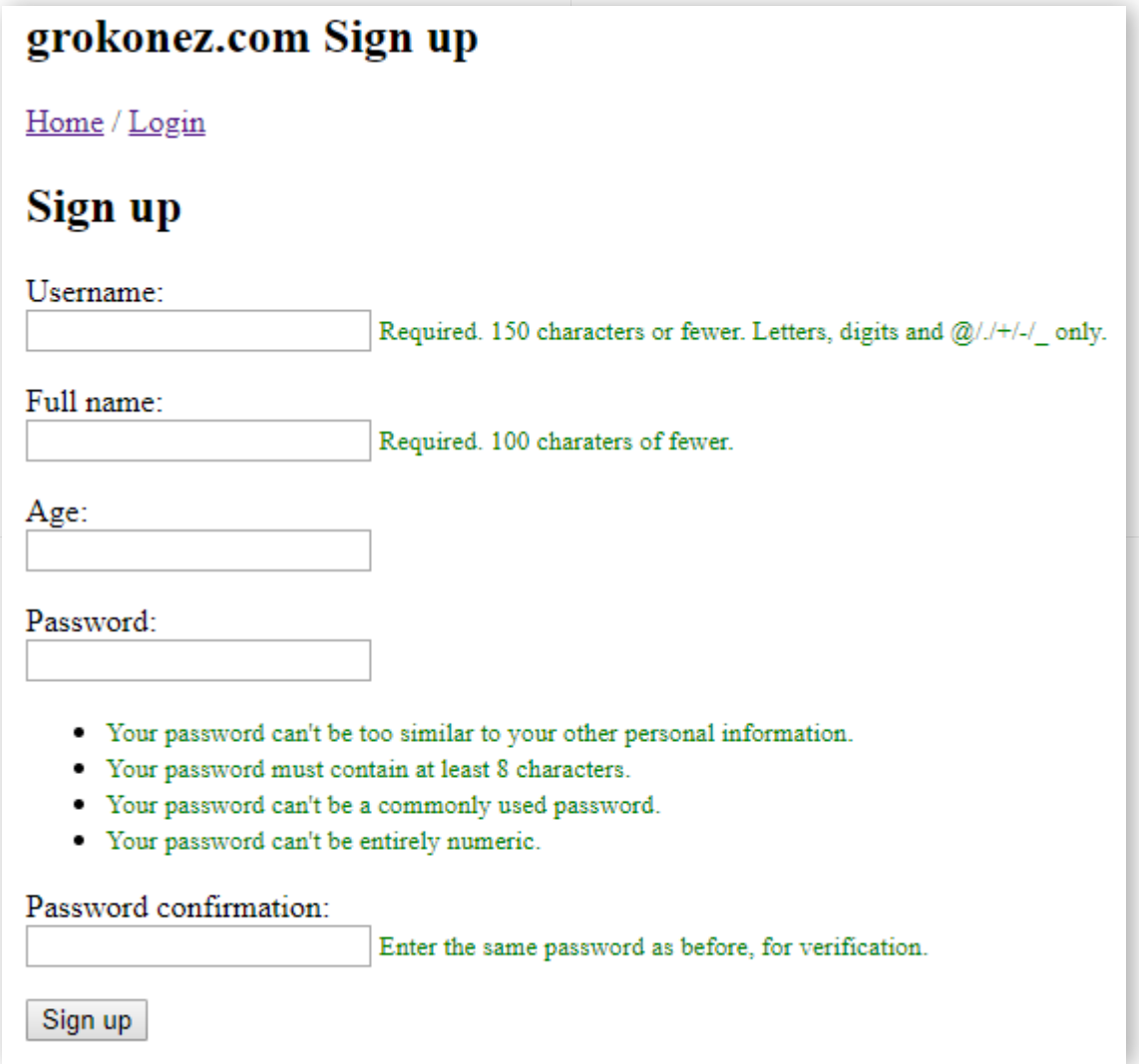
```
<h2>grokonez.com Sign up</h2>

<a href="{% url 'home' %}">Home</a> /
<a href="{% url 'login' %}">Login</a>

<h2>Sign up</h2>
<form method="post">
  {% csrf_token %}
  {% for field in form %}
  <p>
    {{ field.label_tag }}<br/>
    {{ field }}
    {% if field.help_text %}
      <small style="color: green">{{ field.help_text }}</small>
    {% endif %}
    {% for error in field.errors %}
      <p style="color: red">{{ error }}</p>
    {% endfor %}
```

```
  </p>
  {% endfor %}
  <button type="submit">Sign up</button>
</form>
```

It looks like:



Create Django custom Login template

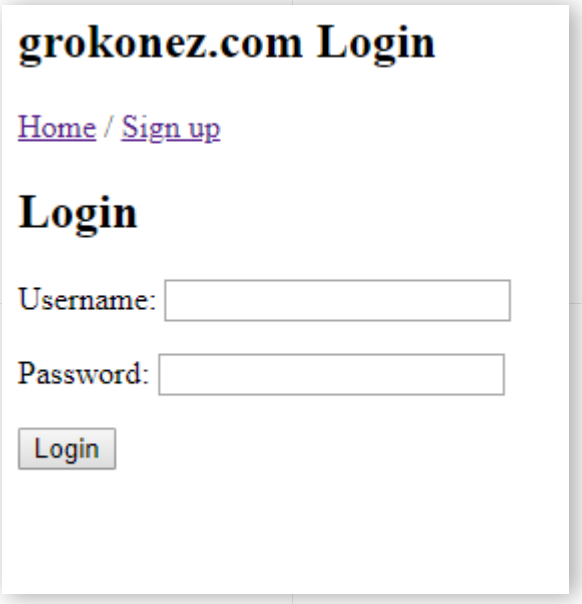
In **templates** folder, create *login.html* file

```
<h2>grokonez.com Login</h2>

<a href="{% url 'home' %}">Home</a> /
<a href="{% url 'signup' %}">Sign up</a>

<h2>Login</h2>
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Login</button>
</form>
```

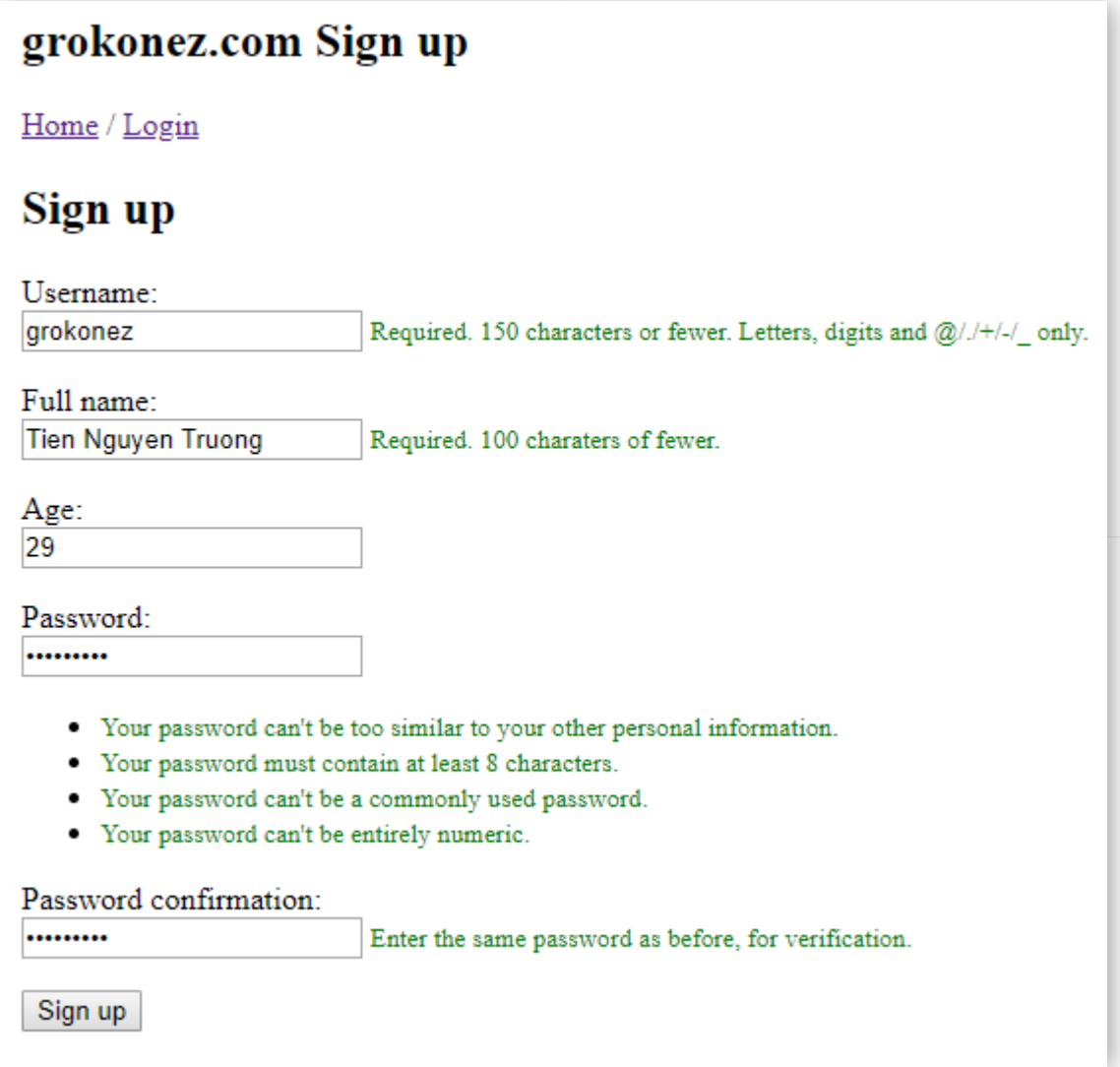
We use `{{ form.as_p }}` to render it within paragraph `<p>` tags.
It looks like:



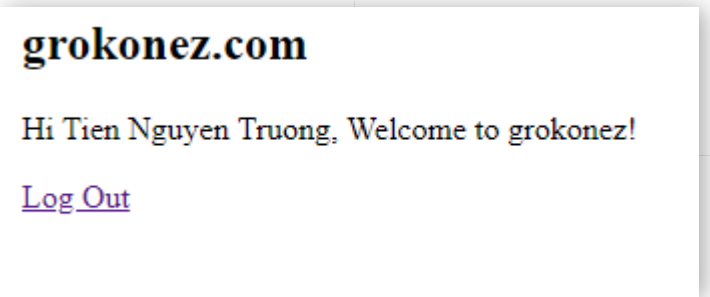
Run & Check results

– Run Django project with command:
`python manage.py runserver`

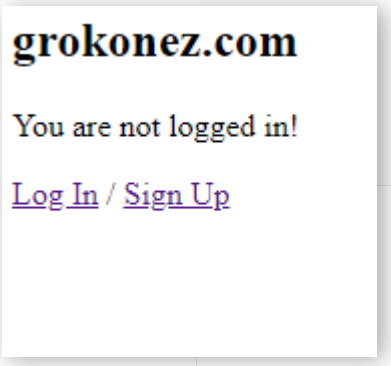
– Open browser with url `http://localhost:8000/`, then go to `/signup` page and fill your information:



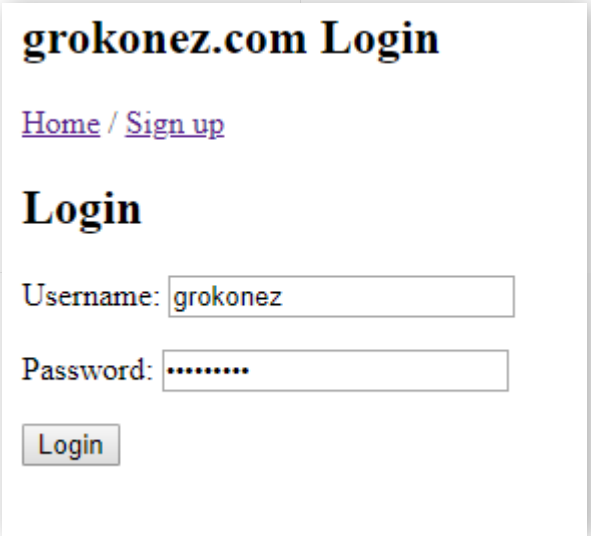
Click on **Sign up** button, if the process is sucessful, the browser will turn into `/home` page with your information:



Click on **Log out** and see the result:



Go to `/login` page and fill *Username* and *Password*:



Click on **Login** to check authentication.

Now check MySQL database:

```
mysql> select * from authen_customuser;
+----+-------------------------------------------------------------------------
| id | password
+----+-------------------------------------------------------------------------
|  1 | pbkdf2_sha256$120000$K7UOPnTpJ6uh$kXJf23sxjj7ft+UGCAKMPF62Fbw06tAc2ArD0Zq/J
+----+-------------------------------------------------------------------------
```

Source Code

DjangoAuth-customUser

Conclusion

Congratulations!

Now you've known how to build a Django Project with Authentication in which, we customize user project instead of using default Django auth User, you also know how to create Custom User Model, how to create a new form for `UserCreationForm`, how to set `urlpatterns` & handle signup/login/logout requests and custom Django template for User Authentication.

Happy learning Django!

By grokonez | March 23, 2019.

Related Posts

- Django RestApis example – GET/POST/PUT/DELETE requests to MongoDb database
- Django RestApis example – GET/POST/PUT/DELETE requests to PostgreSQL database
- Django RestApis example – GET/POST/PUT/DELETE requests MySQL database with Django REST Framework
- Django – How to upload, view, delete file using ModelForm and MySQL
- Django – How to upload file
- Django RestApis CRUD Application with Angular 6 & PostgreSQL tutorial
- Django CRUD Application with VueJs as front-end | VueJs + Django Rest Framework + MySQL example – Part 2: Django Server
- Django CRUD Application with VueJs as front-end | VueJs + Django Rest Framework + MySQL example – Part 1: Overview
- Django + Angular 6 example | Django Rest Framework + MySQL CRUD example – Part 2: Django Server

Post Tags

auth | contri | django | django authentication | django tutorial | login | logout | signup

6 thoughts on "Django Authentication – How to build

Login/Logout/Signup for custom User”

ldoderwbea

July 3, 2020 at 6:07 pm

wbklkswqouamqvlocclcofugbbrkgq

noobie21

July 7, 2020 at 7:21 pm

Thank you so much, it worked perfectly, can you help on how to integrate a bootstrap template into a form ?

blahh

July 7, 2020 at 7:23 pm

Where is the sign up details stored here ?

answerthis !

July 9, 2020 at 5:44 am

What will be the logic for forgot password ?

Pabba

November 5, 2020 at 7:57 am

how password is converting in form of hash in database?

dddusbaorm

November 30, 2020 at 10:40 am

hcetqwmmcmptxjbtznrtstcswoytvs

grokonez

Home | Privacy Policy | Contact Us | Our Team

FOLLOW US

ABOUT US

We are passionate engineers in software development by Java Technology & Spring Framework. We believe that creating little good thing with specific orientation everyday can make great influence on the world someday.