# Django Tutorial Documentation

**Kosmas Sidiropoulos**

**Nov 18, 2019**

# Contents:

CHAPTER 1

# Indices and tables

## 1.1 Introduction

### 1.1.1 The Project - Auction Website

In this django tutorial we are going to create a website that makes auctions. The admin user will be able to add products to the website and create auctions for them. On the other hand the users will be able to bid on items and chat with each other. Also the user can wishlist an auction.

#### Auction and Bidding System

The users will have to buy credits so they can bid. Each credit is a bid. The auction ends after 5 minutes from the last bid.

**Note:** This is a tutorial to create a website not a way to make money.

Let's make an example.

```
The admin creates an auction of a smartphone that has 200€ value.
The auction starts at 0.20€ and ends in 5 minutes.
Now users fight to win the item.
Users are buying credits to bid on the cheap smartphone.
Users spent at least 10€ so they can bid.
Auction goes on.
Finally a user wins the smartphone for 40.20€
The user spent 10€ for the credits and 40.20€ for the auction price.
In this auction 40 users participated (and they also brought credits).
In the end the website spents 200€ for the product but
40 users brought credits 10€ each and the winner spent 40.20€
So the website made profit 40.20€ + (40 * 10€) - 200€ = 240.20€
Everyone is happy.
```

## 1.2 Installation

In this tutorial we will learn how to use the django framework to build a website that makes auctions.

Requirements :

- Python 3.6
- pip 10.0.1
- Django 2.0.2

### 1.2.1 Installing Python 3.6

Download and install python from the official website which you can find here .

### 1.2.2 Installing Pip 10.0.1

To install pip you must first download the get_pip.py which can be found here . Next open your favorite terminal and run the following :

```
>> python get_pip.py
```

### 1.2.3 Installing Django 2.0.2

After pip is installed you can simply type :

```
>> pip install django
```

## 1.3 Creating the project

### 1.3.1 Generating the necessary files

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

```
>> django-admin startproject auctionsonline
```

This will create a directory named auctionsonline in your current directory.

If we look at the directory we created:

```
mysite/
   manage.py
   mysite/
       __init__.py
       settings.py
       urls.py
       wsgi.py
```

This files are:

- `manage.py` : A command-line utility that lets you interact with this Django project in various ways. We will use this file a lot.
- `settings.py` : Settings/configuration for this Django project.
- `urls.py` : The URL declarations for this Django project.
- `__init__.py` : An empty file that tells Python that this directory should be considered a Python package.
- `wsgi.py` : An entry-point for WSGI-compatible web servers to serve your project

### 1.3.2 Testing the Django Project

To verify if the django project works run the following command :

```
>> python manage.py runserver
```

We should see the following output :

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

June 01, 2018 - 15:50:53
Django version 2.0, using settings 'auctionsonline.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

We've started the django development server, a lightweight web server which is used for rapid development. This web server MUSTN'T be used in a production enviroment.

### 1.3.3 Creating the first app

#### Projects vs Apps

This is really more of a separate (though related) question, but understanding the distinction Django draws between a "project" and an "application" is a big part of good code layout. Roughly speaking, this is what the two terms mean:

- An application tries to provide a single, relatively self-contained set of related functions. An application is allowed to define a set of models (though it doesn't have to) and to define and register custom template tags and filters (though, again, it doesn't have to).
- A project is a collection of applications, installed into the same database, and all using the same settings file. In a sense, the defining aspect of a project is that it supplies a settings file which specifies the database to use, the applications to install, and other bits of configuration. A project may correspond to a single web site, but doesn't have to — multiple projects can run on the same site. The project is also responsible for the root URL configuration, though in most cases it's useful to just have that consist of calls to include which pull in URL configurations from inidividual applications.

Views, custom manipulators, custom context processors and most other things Django lets you create can all be defined either at the level of the project or of the application, and where you do that should depend on what's most effective for you; in general, though, they're best placed inside an application (this increases their portability across projects).

### Creating the website app

To create the app we need to be in the same directory as manage.py. Executing this command :

```
>> python manage.py startapp website
```

will create an app called website and the following files will be created.

```
website/
   __init__.py
   admin.py
   apps.py
   migrations/
       __init__.py
   models.py
   tests.py
   views.py
```

This files are:

- `__init__.py` :
- `admin.py` : Registers models to the admin site.
- `apps.py` : This file is created to help the user include any application configuration for the app. Using this, you can configure some of the attributes of the application.
- `migrations/__init__.py` :
- `models.py` : Defines a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection.
- `tests.py` :
- `views.py` : A set of functions that take a web request and return a web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image, etc.

The next step is to include the app in our project, we need to add a reference to its configuration class in the INSTALLED_APPS setting. Edit the auctionsonline/settings.py file and add that dotted path to the INSTALLED_APPS setting. It'll look like this:

```
INSTALLED_APPS = [
   'website',
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
]
```

Now that Django knows to include the website app we are ready to move forward to the next chapter where we will create the database for the website.

## 1.4 The Model Layer

### 1.4.1 Introduction to models

By default, the configuration uses SQLite. If you're new to databases, or you're just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won't need to install anything else to support your database.

### 1.4.2 Field types

- `CharField` : A string field for small strings. Also it requires the maximum length of the string as an argument.

  **Field.choices** : An iterable (e.g., a list or tuple) consisting itself of iterables of exactly two items (e.g. [(A, B), (A, B) . . . ]) to use as choices for this field.

The first element in each tuple is the actual value to be set on the model, and the second element is the human-readable name. For example:

```
CATEGORIES = (
  ('LAP', 'Laptop'),
  ('CON', 'Console'),
  ('GAD', 'Gadget'),
  ('GAM', 'Game'),
  ('TEL', 'TV'))
```

- `DateTimeField` : A date and time, represented in Python.
- `IntegerField` : An integer. Values from -2147483648 to 2147483647 are safe in all databases supported by Django.
- `DecimalField` : A fixed-precision decimal number, represented in Python by a Decimal instance. Has two required arguments

  **DecimalField.max_digits** : The maximum number of digits allowed in the number. Note that this number must be greater than or equal to decimal_places.

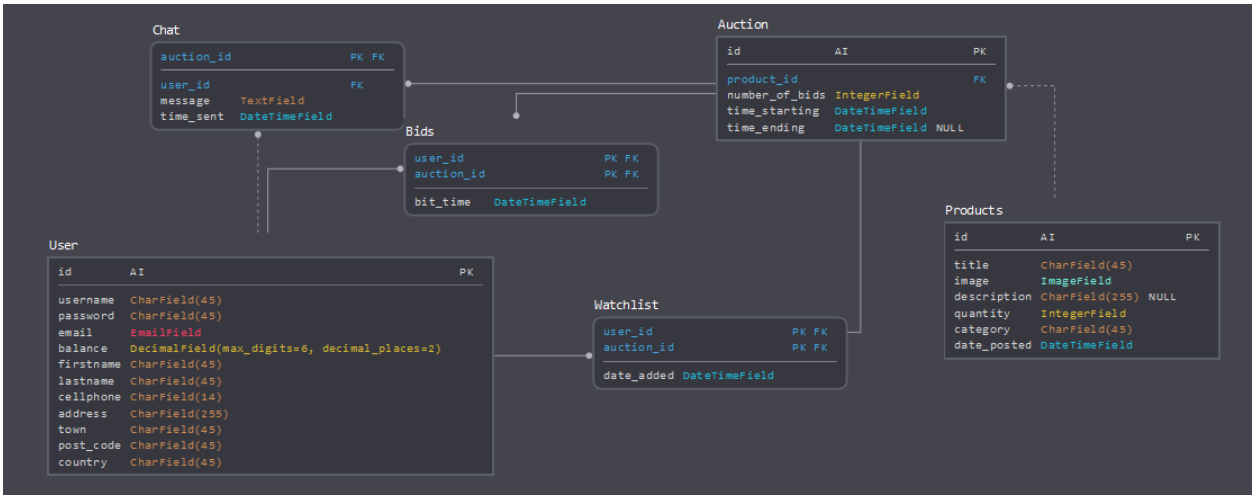
  **DecimalField.decimal_places** : The number of decimal places to store with the number.
- `EmailField` : A CharField that checks that the value is a valid email address. It uses EmailValidator to validate the input.
- `ImageField` : Inherits all attributes and methods from FileField, but also validates that the uploaded object is a valid image.
- `TextField` : A large text field.

For more field types check the official documentation here.

### 1.4.3 Indexes

- `PrimaryKey` : In django an id field is added automatically acting as a primary key. Of course his behavior can be overridden.
- `ForeignKey` : To define a many-to-one relationship, use **django.db.models.ForeignKey**. You use it just like any other Field type: by including it as a class attribute of your model.

### 1.4.4 Building our own models




The `models.py` file must look like this :

```
from django.db import models

# Create your models here.
class User(models.Model):
        username = models.CharField(max_length=45)
        password = models.CharField(max_length=45)
        email = models.EmailField()
        balance = models.DecimalField(max_digits=6, decimal_places=2)
        firstname = models.CharField(max_length=56)
        lastname = models.CharField(max_length=45)
        cellphone = models.CharField(max_length=14)
        address = models.CharField(max_length=255)
        town = models.CharField(max_length=45)
        post_code = models.CharField(max_length=45)
        country = models.CharField(max_length=45)

class Product(models.Model):
        CATEGORIES = (
            ('LAP', 'Laptop'),
            ('CON', 'Console'),
            ('GAD', 'Gadget'),
            ('GAM', 'Game'),
            ('TEL', 'TV')
        )

        title = models.CharField(max_length=255)
        image = models.ImageField()
        description = models.CharField(max_length = 500)
        quantity = models.IntegerField()
        category = models.CharField(
            max_length=2,
            choices=CATEGORIES
        )
        date_posted = models.DateTimeField(auto_now_add=True, blank=True)

class Auction(models.Model):
```

(continues on next page)

(continued from previous page)

```
        product_id = models.ForeignKey(Product, on_delete=models.CASCADE)
        number_of_bids = models.IntegerField()
        time_starting = models.DateTimeField()
        time_ending = models.DateTimeField()

class Watchlist(models.Model):
        user_id = models.ForeignKey(User, on_delete=models.CASCADE)
        auction_id = models.ForeignKey(Auction, on_delete=models.CASCADE)

class Bid(models.Model):
        user_id = models.ForeignKey(User, on_delete=models.CASCADE)
        auction_id = models.ForeignKey(Auction, on_delete=models.CASCADE)
        bid_time = models.DateTimeField()

class Chat(models.Model):
        auction_id = models.ForeignKey(Auction, on_delete=models.CASCADE)
        user_id = models.ForeignKey(User, on_delete=models.CASCADE)
        message = models.TextField()
        time_sent = models.DateTimeField()
```

### 1.4.5 Adding our models to the database

Now that we've written our models we have to migrate them into our database. We can achive this by running the following commands:

```
>> python manage.py makemigrations website
```

You should see something similar to the following:

```
Migrations for 'website':
website\migrations\0001_initial.py
  - Create model Auction
  - Create model Bid
  - Create model Chat
  - Create model Product
  - Create model User
  - Create model Watchlist
  - Add field user_id to chat
  - Add field user_id to bid
  - Add field product_id to auction
```

By running makemigrations, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a migration. Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. You can read the migration for your new model if you like it's the file website/migrations/0001_initial.py.

There's a command that will run the migrations for you and manage your database schema automatically - that's called migrate, and we'll come to it in a moment - but first, let's see what SQL that migration would run. The sqlmigrate command takes migration names and returns their SQL:

```
>> python manage.py sqlmigrate website 0001
```

You should see something similar to the following:

```
BEGIN;
--
-- Create model Auction
--
CREATE TABLE "website_auction" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
↪"number_o
f_bids" integer NOT NULL, "time_starting" datetime NOT NULL, "time_ending" datetime␣
↪NOT NU
LL);
--
-- Create model Bid
--
CREATE TABLE "website_bid" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "bid_time
↪" da
tetime NOT NULL, "auction_id_id" integer NOT NULL REFERENCES "website_auction" ("id")␣
↪DEFE
RRABLE INITIALLY DEFERRED);
--
-- Create model Chat
--
CREATE TABLE "website_chat" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "message
↪" te
xt NOT NULL, "time_sent" datetime NOT NULL, "auction_id_id" integer NOT NULL␣
↪REFERENCES "w
ebsite_auction" ("id") DEFERRABLE INITIALLY DEFERRED);
--
-- Create model Product
--
CREATE TABLE "website_product" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
↪"title" v
archar(255) NOT NULL, "image" varchar(100) NOT NULL, "description" varchar(500) NOT␣
↪NULL,
"quantity" integer NOT NULL, "category" varchar(2) NOT NULL, "date_posted" datetime␣
↪NOT NU
LL);
--
-- Create model User
--
CREATE TABLE "website_user" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
↪"username" v
archar(45) NOT NULL, "password" varchar(45) NOT NULL, "email" varchar(254) NOT NULL,
↪"bala
nce" decimal NOT NULL, "firstname" varchar(56) NOT NULL, "lastname" varchar(45) NOT␣
↪NULL,
"cellphone" varchar(14) NOT NULL, "address" varchar(255) NOT NULL, "town" varchar(45)␣
↪NOT
NULL, "post_code" varchar(45) NOT NULL, "country" varchar(45) NOT NULL);
--
-- Create model Watchlist
--
CREATE TABLE "website_watchlist" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
↪"auctio
n_id_id" integer NOT NULL REFERENCES "website_auction" ("id") DEFERRABLE INITIALLY␣
↪DEFERRE
D, "user_id_id" integer NOT NULL REFERENCES "website_user" ("id") DEFERRABLE␣
↪INITIALLY DEF
ERRED);
```

(continues on next page)

(continued from previous page)

```
--
-- Add field user_id to chat
--
ALTER TABLE "website_chat" RENAME TO "website_chat__old";
CREATE TABLE "website_chat" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "message
↪" te
xt NOT NULL, "time_sent" datetime NOT NULL, "auction_id_id" integer NOT NULL␣
↪REFERENCES "w
ebsite_auction" ("id") DEFERRABLE INITIALLY DEFERRED, "user_id_id" integer NOT NULL␣
↪REFERE
NCES "website_user" ("id") DEFERRABLE INITIALLY DEFERRED);
INSERT INTO "website_chat" ("id", "message", "time_sent", "auction_id_id", "user_id_id
↪") S
ELECT "id", "message", "time_sent", "auction_id_id", NULL FROM "website_chat__old";
DROP TABLE "website_chat__old";
CREATE INDEX "website_bid_auction_id_id_8a24134d" ON "website_bid" ("auction_id_id");
CREATE INDEX "website_watchlist_auction_id_id_1ce8deb1" ON "website_watchlist" (
↪"auction_i
d_id");
CREATE INDEX "website_watchlist_user_id_id_517566fa" ON "website_watchlist" ("user_id_
↪id")
;
CREATE INDEX "website_chat_auction_id_id_17d789bb" ON "website_chat" ("auction_id_id
↪");
CREATE INDEX "website_chat_user_id_id_66161742" ON "website_chat" ("user_id_id");
--
-- Add field user_id to bid
--
ALTER TABLE "website_bid" RENAME TO "website_bid__old";
CREATE TABLE "website_bid" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "bid_time
↪" da
tetime NOT NULL, "auction_id_id" integer NOT NULL REFERENCES "website_auction" ("id")␣
↪DEFE
RRABLE INITIALLY DEFERRED, "user_id_id" integer NOT NULL REFERENCES "website_user" (
↪"id")
DEFERRABLE INITIALLY DEFERRED);
INSERT INTO "website_bid" ("id", "bid_time", "auction_id_id", "user_id_id") SELECT "id
↪", "
bid_time", "auction_id_id", NULL FROM "website_bid__old";
DROP TABLE "website_bid__old";
CREATE INDEX "website_bid_auction_id_id_8a24134d" ON "website_bid" ("auction_id_id");
CREATE INDEX "website_bid_user_id_id_7cc0c150" ON "website_bid" ("user_id_id");
--
-- Add field product_id to auction
--
ALTER TABLE "website_auction" RENAME TO "website_auction__old";
CREATE TABLE "website_auction" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
↪"number_o
f_bids" integer NOT NULL, "time_starting" datetime NOT NULL, "time_ending" datetime␣
↪NOT NU
LL, "product_id_id" integer NOT NULL REFERENCES "website_product" ("id") DEFERRABLE␣
↪INITIA
LLY DEFERRED);
INSERT INTO "website_auction" ("id", "number_of_bids", "time_starting", "time_ending",
↪ "pr
oduct_id_id") SELECT "id", "number_of_bids", "time_starting", "time_ending", NULL␣
↪FROM "we
```

(continues on next page)

(continued from previous page)

```
bsite_auction__old";
DROP TABLE "website_auction__old";
CREATE INDEX "website_auction_product_id_id_b4d0e759" ON "website_auction" ("product_
↪id_id
");
COMMIT;
```

Now, run migrate again to create those model tables in your database:

```
>> python manage.py migrate
```

we should see the following:

```
Operations to perform:
       Apply all migrations: admin, auth, contenttypes, sessions, website
Running migrations:
       Applying contenttypes.0001_initial... OK
       Applying auth.0001_initial... OK
       Applying admin.0001_initial... OK
       Applying admin.0002_logentry_remove_auto_add... OK
       Applying contenttypes.0002_remove_content_type_name... 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 sessions.0001_initial... OK
       Applying website.0001_initial... OK
```

The migrate command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called django_migrations) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

### 1.4.6 Shell

Now, we can use an interactive Python shell that API Django gives you. To invoke the Python shell, use this command:

```
>> $ python manage.py shell
```

We should see the following text:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on␣
↪win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Using the shell we will create a new user for our website. Running the following commands:

```
>>> from website.models import User
>>> # Creates an User object.
>>> user1 = User()
```

(continues on next page)

(continued from previous page)

```
>>> user1.username = "dummy1"
>>> user1.email = "dummy1@mail.com"
>>> user1.password = "dummypassword"
>>> user1.balance = 20.0
>>> user1.firstname = "Dummy"
>>> user1.lastname = "One"
>>> user1.cellphone = "6988757575"
>>> user1.address = "Dumadd 199"
>>> user1.town = "Dummtown"
>>> user1.post_code = "35100"
>>> user1.country = "Dummcon"
>>> # Saves User object to the database.
>>> user1.save()
```

If we want to check if the user was successfully registered we execute this command:

```
>>> User.objects.all()
<QuerySet [<User: User object (1)>]>
```

The result we get is quite unclear. Lets fix it by opening the models.py file and adding a **__str__()** method to User class

```
class User(models.Model):
    ...

    def __str__(self):
        return "(" + self.username + ", " + self.email + ")"
```

Now lets execute again the previous command:

```
>>> User.objects.all()
<QuerySet [<User: (dummy1, dummy1@mail.com)>]>
```

The **User.objects.all()** is displaying all the User records in the database showing the username and the email of each user.

### 1.4.7 Retrieving specific objects with filters

The QuerySet returned by *all()* describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects. To create such a subset, you refine the initial QuerySet, adding filter conditions.

One way to do this is with the:

- **filter(**kwargs)**

  Returns a new QuerySet containing objects that match the given lookup parameters.

for example if we want to find the user with 'dummy1@mail.com' we will use:

```
>>> User.objects.filter(email='dummy1@mail.com')
<QuerySet [<User: ID:1 dummy1 dummy1@mail.com>]>
```

One more thing we will need from the QuerySet API are the field lookups. Field lookups are how you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the QuerySet methods

for example if we want to find users with id greater than 5 we will use the *gt* field lookup :

```
>>>User.objects.filter(id__gt=5)
<QuerySet []>
```

Some usefull field lookups are:

- `gt` : Greater than.
- `lt` : Less than
- `gte` : Greater than or equal to.
- `lte` : Less than or equal to.

## 1.5 Introducing the Django Admin

### 1.5.1 Creating an admin user

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

```
>> python manage.py createsuperuser
```

Enter your desired username and press enter.

```
>> Username: admin
```

You will then be prompted for your desired email address:

```
>> Email address: admin@admin.com
```

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

```
>> Password: **********
>> Password (again): *********
>> Superuser created successfully.
```

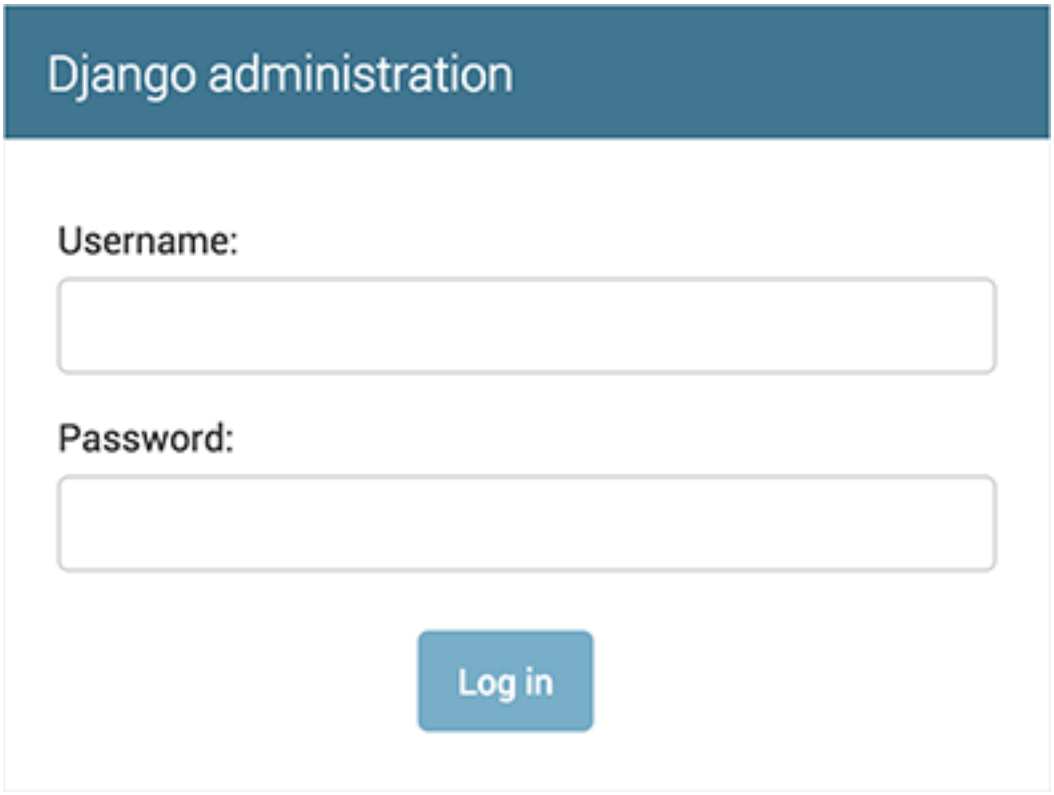
### 1.5.2 Start the development server

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

If the server is not running start it like so:

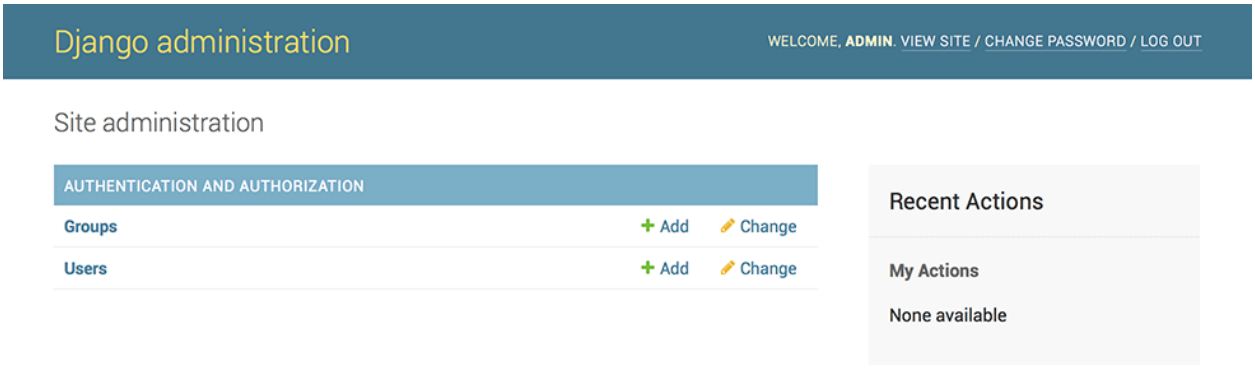
```
>> python manage.py runserver
```

Now, open a Web browser and go to "127.0.0.1:8000/admin/". You should see the admin's login screen:




### 1.5.3 Enter the admin site

Now, try logging in with the superuser account you created in the previous step. You should see the Django admin index page:




You should see a few types of editable content: groups and users. They are provided by django.contrib.auth, the authentication framework shipped by Django. We will not use those in this tutorial.

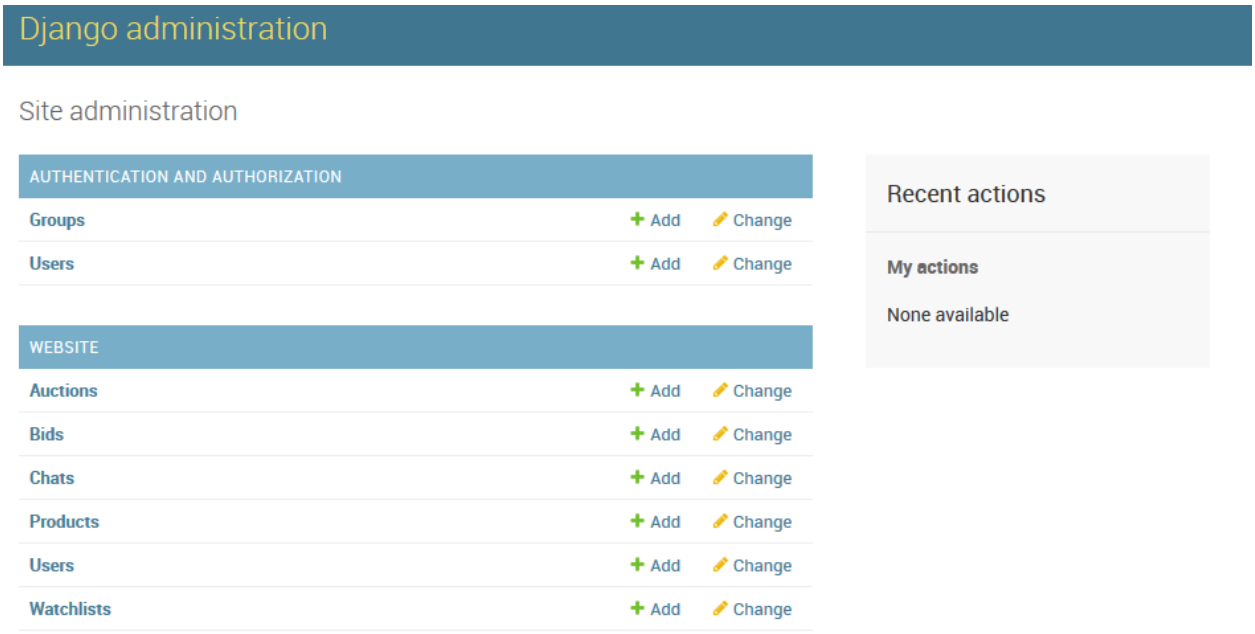
### 1.5.4 Make the website app modifiable in the admin

To do this, open the website/admin.py file, and edit it to look like this:

```
from django.contrib import admin
from .models import User, Product, Auction, Chat, Watchlist, Bid

# Register your models here.
admin.site.register(User)
admin.site.register(Product)
admin.site.register(Auction)
admin.site.register(Chat)
admin.site.register(Watchlist)
admin.site.register(Bid)
```

If we refresh the admin site we should se our models.




From the admin website we can manage the database records. We can edit, delete and add records.

## 1.6 The View Layer

"Django has the concept of "views" to encapsulate the logic responsible for processing a user's request and for returning the response."

To begin, in the primary directory of your project (for us the "auctionsonline"), there is a python file called "urls.py". In order to be able to connect the urls we will later create for our website, we need to include them to the specific file, as well as the appropriate libraries. Open urls.py and edit it:

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

urlpatterns = [
   path('admin/', admin.site.urls),
   path('website/', include('website.urls'))
]
```

Now going to create a "urls.py" file on our app directory ("website"), we should now be able to create the paths for our website pages. We need to import the views from the "website" directory so as to connect our path with the appropriate "response", which we will later create.

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

app_name = 'website'

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

### 1.6.1 URLs

Here's our URLconf:

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

urlpatterns = [
    path('', views.index, name='index'),
    path('login/', views.login_page, name='login_view'),
    path('logout/', views.logout_page, name='logout_view'),
    path('register/', views.register_page, name='register_page'),
    path('register/new_user/', views.register, name='register'),
    path('category/<str:category>/', views.filter_auctions, name='filter_auctions'),
    path('watchlist/<int:auction_id>/', views.watchlist, name='watchlist'),
    path('balance/', views.balance, name='balance'),
    path('balance/topup/', views.topup, name='topup'),
    path('watchlist/', views.watchlist_page, name='watchlist'),
    path('bid/<int:auction_id>/', views.bid_page, name='bid_page'),
    path('bid/<int:auction_id>/comment/', views.comment, name='comment'),
    path('bid/<int:auction_id>/raise_bid/', views.raise_bid, name='raise_bid'),
]
```

Notes:

To capture a value from the URL, use angle brackets. Captured values can optionally include a converter type. For example, use <int:auction_id> to capture an integer parameter. If a converter isn't included, any string, excluding a / character, is matched. There's no need to add a leading slash, because every URL has that. For example, it's balance, not /balance.

Example requests:

```
A request to bid/34/comment/ would match the twelfth entry in the list.
Django would call the function views.comment(request, auction_id)(request, auction_
↪id).

category/laptops would match the sixth pattern in the list.

bid/34/comment/ would match the second from last pattern in the list. The patterns␣
↪are tested in order.
```

### Path converters

The following path converters are available by default:

- `str` : Matches any non-empty string, excluding the path separator, '/'. This is the default if a converter isn't included in the expression.
- `int` : Matches zero or any positive integer. Returns an int.
- `slug` : Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, building-your-1st-django-site.
- `uuid` : Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, 075194d3-6885-417e-a8a8-6c931e272f00. Returns a UUID instance.
- `path` : Matches any non-empty string, including the path separator, '/'. This allows you to match against a complete URL path rather than just a segment of a URL path as with str.

After all our paths are created, we go to the "views.py" file in the same directory, so as to connect the user request with the appropriate html response.

```
....
# Create your views here.
def index(request):
    ....
    return render(request, 'index.html', {'auctions': auctions})

def bid_page(request, auction_id):
    ....
    return index(request)

def comment(request, auction_id):
    ....
    return index(request)

def raise_bid(request, auction_id):
    ....
    return bid_page(request, auction_id)

def register_page(request):
    ....
    return render(request, 'register.html')

def watchlist(request, auction_id):
    ....
    return index(request)

def watchlist_page(request):
    ....
    return index(request)

def balance(request):
    ....
    return index(request)

def topup(request):
    ....
    return index(request)
```

(continues on next page)

(continued from previous page)

```
def filter_auctions(request, category):
    ....
    return index(request)

def register(request):
    ....
    return index(request)

def login_page(request):
    ....
    return index(request)

def logout_page(request):
    ....
    return index(request)
```

So now if we type to our browser "127.0.0.1:8000/website/" we request from the server the index.html file. Depending the url the equivalent view is called.

### 1.6.2 render()

*render(request, template_name, context=None, content_type=None, status=None, using=None)[source]*

Combines a given template with a given context dictionary and returns an HttpResponse object with that rendered text. Django does not provide a shortcut function which returns a TemplateResponse because the constructor of TemplateResponse offers the same level of convenience as render().

Required arguments:

- `request` : The request object used to generate this response.
- `template_name` : The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. See the template loading documentation for more information on how templates are found.

Optional arguments:

- `context` : We can give dictionary mapping variable names to variable values.

## 1.7 The Template Layer

"The template layer provides a designer-friendly syntax for rendering the information to be presented to the user."

Django has a built in function to search by default for html files and templates in a directory named "templates", so we create one in the app directory, and we store all the html files for the project

```
website/
    templates/
        balance.html
        bid.html
        home.html
        index.html
        products.html
        register.html
```

(continues on next page)

(continued from previous page)

```
admin.py
apps.py
models.py
tests.py
urls.py
views.py
```

Whenever we want to inject ,in a specific location, a specific file or function, action or url, we can use specific django formats :

### 1.7.1 Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by `{{` and `}}` like this:

My name is `{{ user.name }}`

### 1.7.2 Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by `{%` and `%}` like this:

`{% extends "base_template.html" %}` : indicates that the specific template/part of code extends another template/part of code.

`{% include "some_template.html" %}` : indicates that in the specific part of the code, another template/code will be included.

`{% block label %}` block `{% endblock %}` : we use this format to enclose parts of the code we want to be replaced or included, when a specific request/event is triggered.

**if/else if/else statements**

```
{% if 'statement' %}
    ...code...
{% elif %}
    ...code...
{% else %}
    ...code...
{% endif %}
```

**For loops**

```
{% for ... %}
    ...code...
{% endfor %}
```

`{% url 'website:index' %}` : page redirect (views name)

`{% now "Y-m-d-H-i-s" as todays_date %}` : declares a new variable that holds the current date/time.

### 1.7.3 Static files

Websites generally need to serve additional files such as images, JavaScript, or CSS. In Django, we refer to these files as "static files".

Create a new directory called 'static' in the app directory

```
website/
   migrations/
   templates/
   static/
       css/ <- Here we save the css files
       images/ <- here the images
```

Now that we added the static files we also have to make a directory for the file the admin uploads (for creating auctions, the item picture).

Open the settings.py and add the following lines to the end.

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

also open the urls.py and edit the code so it will look like this:

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

### 1.7.4 Tag Filters

Filters the contents of the block through one or more filters. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax.

```
{% if messages|length >= 100 %}
  You have lots of messages today!
{% endif %}
```

the length filter returns the length of the value. This works for both strings and lists. There are many filters in the Django documentation, however there are occasions where we have to calculate something more complex than the given filters. Django gives us the ability to create custom filters.

### 1.7.5 Creating custom filters

To create custom filters we have to create a directory called 'templatetags' in the our app and inside it we will create two files.

- __init__.py
- custom_tags.py : Here we will define filters we will use on our templates

The custom_tags.py will look like this:

```
from django import template
from ..models import User
from django.utils import timezone

register = template.Library()

@register.filter(name='search')
def search(value, id):
    for v in value:
        if v.id == id:
            return True

    return False

@register.filter(name="time_left")
def time_left(value):
    t = value - timezone.now()
    days, seconds = t.days, t.seconds
    hours = days * 24 + seconds // 3600
    minutes = (seconds % 3600) // 60
    seconds = seconds % 60
    st = str(minutes) + "m " + str(seconds) + "s"
    return st

@register.filter(name="current_price")
def current_price(value):
    current_cost = 0.20 + (value.number_of_bids * 0.20)
    current_cost = "%0.2f" % current_cost
    return current_cost
```

We just created three new filters

- search : searches a list for a key

```
{% if watchlist|search:auction.id%} watchlist contains auction_id {% endif %}
```

- time_left : string representation of the auction's time left

```
{{ auction.time_ending|time_left }}
```

- current_price : the current price of the auction item based on the number of bids

```
€{{ auction|current_price }}
```

### 1.7.6 Creating the project's templates

#### The Index Template

For our project, the primary template is "index.html", which contains all the necessary code for our site's appearance. The index page contains a navbar with its components, a dummy ad, a footer, a login modal and finally the "replacement-container" which is altered depending the content we want to show.

The replacement-container :

```
<div class="p-2">
    <div id="replacement">
```

(continues on next page)

(continued from previous page)

```
        {% block body %}
        {% include "products.html" %}
        {% endblock %}
    </div>
</div>
```

Whenever we are requesting the index page by default the replacement container loads the contents of the products.html

If the user is currently logged in

```
<!-- If the user is logged in then include the home.html contents in the navbar -->
{% if request.session.username %}
    {% include "home.html" %}
<!-- else add login and register links -->
{% else %}
    <li class="nav-item open-modal">
        <a id="login" href="#myModal" class="nav-link trigger-btn" data-toggle="modal
↪">Login</a>
    </li>
    <li class="nav-item">
        <a id="signup" class="nav-link" href="/website/register/">Sign up</a>
    </li>
{% endif %}
```

After the footer we have a modal with a login form. We will talk on how to handle forms in the next section.

### The Product Template

The products template contains the auctions that are still in process and all the other auctions that will start in the future.

In this template we are going to need the custom tags we made

```
{% load custom_tags %}
```

So the product template will display all the auctions by running a loop on the auction QuerySet. Each auction will be displayed as a card and depending the starting_time and ending time the auction will be active (we will be able to bid) or inactive (just showing the starting_time). Also one more QuerySet is given as a parameter the watchlist QuerySet. On every action loop we are going to check if the auction_id exists also in the watchlist QuerySet. We are achiving this with the custom tag we created 'search'.

```
<!-- If there is a watchlist parameter ... -->
{% if watchlist %}
    <!-- If the auction id is also in the watchlist QuerySet ... -->
    {% if watchlist|search:auction.id%}
        Unwatch
    {% else %}
        Watch
    {% endif %}
{% else %}
    Watch
{% endif %}
```

#### The Bid Template

The bid.html contains the page that allows users to bid on auctions. It also has a static chat for the users to communicate with each other.

The user will be able to add or remove the auction to the watchlist.

In this template we have four parameters from the view model the auction and user information one special list that contains data we created inside the bid_page view and the user's watchlist.

The static chat is a form that makes a POST request. We will talk on how to handle forms in the next section.

#### The Resister Template

The register template contains a form that makes a POST request. We will talk on how to handle forms in the next section.

## 1.8 Forms

Django provides a rich framework to facilitate the creation of forms and the manipulation of form-data. In our main directory (website) we create a python file, named "forms.py" were we place the code for the forms we require.

### 1.8.1 Project's Forms

---

**Note:** The Form's variables we are going to declare must have the same name with the <input ... name="username" ... />

---

#### The Register form

To access the services of our website, the user must register by completing the form presented when clicking the "sign up" button in the nav-bar. When pressed, the form fields will replace the products shown in the index page. To create the form, we access the "forms.py" file and write down the required code and the fields we want the user to fill in :

```
from django import forms

class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=45)
    email = forms.EmailField()
    password1 = forms.CharField(max_length=45)
    password2 = forms.CharField(max_length=45)
    .....
```

#### The Login Form

In order to login in our website, one must complete the required credentials of the login form, which are contained in the "login modal" that is activated when the user presses the "sign in" button in the navbar. To create the form, we access the "forms.py" file and write down the required code and the fields we want the user to fill in :

```
....
class LoginForm(forms.Form):
    username = forms.CharField(max_length=45)
    password = forms.CharField(max_length=45)
    ....
```

#### General Rule

- Create the form class with the form fields you require
- In the template file don't forget to give the same name to the form elements
- In the views.py when you want to handle forms use the following code:

```
if request.method == 'POST':
    form = RegistrationForm(request.POST)
    if form.is_valid():
```

- And finally to get a field value use:

```
form.cleaned_data[...]
```

### 1.8.2 The CSRF Token

"The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered. The first defense against CSRF attacks is to ensure that GET requests are side effect free. In any template that uses a POST form, use the csrf_token tag inside the <form> element if the form is for an internal URL, e.g.

```
....
<form method="post" ...>
    {% csrf_token %}
    .....
</form>
```

This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.

### 1.8.3 Handling Forms

When the user makes a POST request a view function is called and by typing the following code:

```
if request.method == 'POST':
    form = RegistrationForm(request.POST)
    if form.is_valid():
```

we can process the given data anyway we want. Most of the time is to make a query on our database.

To get each POST parameter we must use:

```
form.cleaned_data[form_field_name]
```

This data has not only been validated but will also be converted in to the relevant Python types for you.

## 1.9 Testing our Website

We've created for you a file called 'populate.py' which is located inside the website app.

`populate.py`: Removes everything from the database and generates User, Product and Auction records in the database.

The first auction from the generated records starts in 1 minute.