Bootcamp Training

Django

Authorized & published by Summitworks Technologies Inc

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Django – Day 10

- **Database Models and ORM Configuration**
  - Django MVT Pattern
  - Configuring Django for Database Access
  - Defining Django Models
  - Understanding Model Fields & Options
  - Table Naming Conventions
  - Creating A Django Model
  - Adding the App to Your Project
  - Generating & Reviewing the SQL
  - Adding Data to the Model
  - Primary Keys and the Model
  - Simple Data Retrieval Using a Model
  - Migrations
  - Understanding QuerySets
  - Applying Filters
  - Specifying Field Lookups
  - Slicing QuerySets
  - Specifying Ordering in QuerySets
  - Common QuerySet Methods
  - Deleting Records
  - Managing and Retrieving related records
  - Retrieving Related Records

- **Database Models and ORM Configuration**
  - Creating Forms from Models
  - Deploy in Windows IIS server
  - Deploy in Apache Server
- **Admin Interface**
  - Creating super user
  - Adding models to admin
  - Customizing Admin interface
- **Static files**
  - Loading css files into templates
  - Loading js files into templates
  - Uploading image using models
  - User authentication
  - Internationalization
- **Django's Cache Framework**
  - Memcached
  - Database Caching
  - Filesystem Caching
  - Local-Memory Caching
  - Dummy Caching (For Development)
  - Template Fragment Caching

@ Copyright 2019, Summitworks Technologies Inc.

Introduction

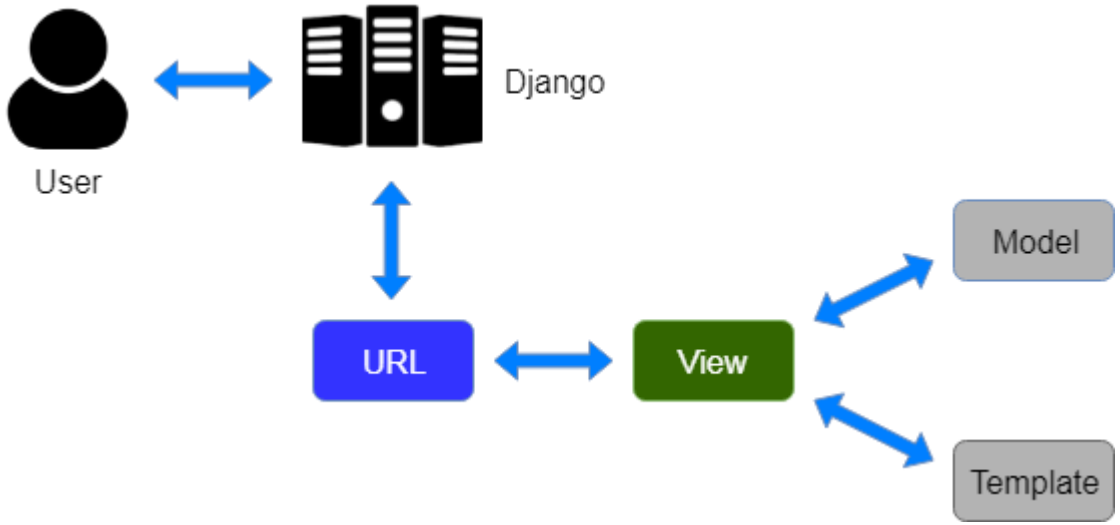
@ Copyright 2019, Summitworks Technologies Inc.

Django MVT

- MVT (Model View Template) is a software design pattern which is a collection of three parts Model View and Template. The Model helps to handle database. It is a data access layer that handles the data.
- The Template is a presentation layer that handles all the User Interface part. The View is used to execute the business logic and interact with model to carry data and renders template.
- Although Django follows MVC pattern but maintains it’s own conventions so controlling is handle by the framework itself.
- There is no separate controller and complete application is based on Model View and Template. That’s why it is called MVT application.

@ Copyright 2019, Summitworks Technologies Inc.

Django MVT



Here, a user **requests** for a resource to the Django, Django works as a controller and check to the available resource in url.
If url maps, **view is called** that interact with model and template, it renders a template.
Django respond back to the user and send a template as **response**

@ Copyright 2019, Summitworks Technologies Inc.

Configuring Django for Database Access

- In addition to SQLite, Django also has support for other popular databases that include: PostgreSQL, MySQL and Oracle. The Django configuration to connect to a database is done inside the settting.py file of a Django project in the DATABASES variable.
- If you open the settings.py file of a Django project you'll notice the DATABASES variable has a default Python dictionary with the values

Default Django DATABASES dictionary

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django ENGINE value for different databases

Django takes care of this issue by supporting different backends or engines. Therefore, depending on the database brand you plan to use for a Django application, the ENGINE value has to be one of the values from below table

Database	Django ENGINE value
MySQL	django.db.backends.mysql
Oracle	django.db.backends.oracle
PostgreSQL	django.db.backends.postgresql_psycopg2
SQLite	django.db.backends.sqlite3

The Django database connection parameter NAME is used to identify a database instance, and its value convention can vary depending on the database brand. For example, for SQLite NAME indicates the location of a flat file, whereas for MySQL it indicates the logical name of an instance.

@ Copyright 2019, Summitworks Technologies Inc.

Defining Django Models

- A Django model is a description of the data in your database, represented as Python code. It's your data layout – the equivalent of your SQL CREATE TABLE statements – except it's in Python instead of SQL, and it includes more than just database column definitions.

- Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables. Django also uses models to represent higher-level concepts that SQL can't necessarily handle.

@ Copyright 2019, Summitworks Technologies Inc.

Your First Model

- I'll focus on a basic book/author/publisher data layout. I use this as our example because the conceptual relationships between books, authors, and publishers are well known, and this is a common data layout used in introductory SQL textbooks. You're also reading a book that was written by an author and produced by a publisher!
- I'll suppose the following concepts, fields, and relationships:
  - An author has a first name, a last name and an email address.
  - A publisher has a name, a street address, a city, a state/province, a country, and a web site.
  - A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship – aka foreign key – to publishers).

@ Copyright 2019, Summitworks Technologies Inc.

Your First Model

```
from django.db import models
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

@ Copyright 2019, Summitworks Technologies Inc.

Understanding Model Fields & Options

Field name	meaning
AutoField	An IntegerField that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify otherwise. See Automatic primary key fields.
BigAutoField	A 64-bit integer, much like an AutoField except that it is guaranteed to fit numbers from 1 to 9223372036854775807.
BigIntegerField	A 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to9223372036854775807. The default form widget for this field is a TextInput.
BinaryField	A field to store raw binary data. It only supports bytes assignment. Be aware that this field has limited functionality. For example, it is not possible to filter a queryset on a BinaryField value.
BooleanField	The default form widget for this field is CheckboxInput, or NullBooleanSelect if null=True.
CharField	The maximum length (in characters) of the field. The max_length is enforced at the database level and in Django's validation using MaxLengthValidator.
DateField	A date, represented in Python by a datetime.date instance. Has a few extra, optional arguments:
DateTimeField	A date and time, represented in Python by a datetime.datetime instance. Takes the same extra arguments as DateField.
DecimalField	A fixed-precision decimal number, represented in Python by a Decimal instance. It validates the input usingDecimalValidator.
DurationField	A field for storing periods of time - modeled in Python by timedelta. When used on PostgreSQL, the data type used is an interval and on Oracle the data type is INTERVAL DAY(9) TO SECOND(6). Otherwise a bigint of microseconds is used.

@ Copyright 2019, Summitworks Technologies Inc.

Table Naming Conventions

- The default convention is better and cleaner to use :

    - It avoids any table naming conflict (As It's a combination of App name and Model name)
    - It creates well organized database (Tables are ordered by App names)

So until you have any special case that needs special naming convention , use the default.

@ Copyright 2019, Summitworks Technologies Inc.

Creating A Django Model

- A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.
- The basics:
  - Each model is a Python class that subclasses django.db.models.Model.
  - Each attribute of the model represents a database field.
  - With all of this, Django gives you an automatically-generated database-access API

Quick example:
This example model defines a Person, which has a first_name and last_name:

```
from django.db import models
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

first_name and last_name are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

@ Copyright 2019, Summitworks Technologies Inc.

Adding the App to Your Project

- To create a application :
  - python manage.py startapp application_name [in command promt]
  - Update settings.py file
  - Update models.py
  - Migrate the models

```
INSTALLED_APPS = (
            'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.sites',
   'django.contrib.messages',
   'django.contrib.staticfiles',
            ‘application_name‘
,)
```

@ Copyright 2019, Summitworks Technologies Inc.

Migrations

- **$ python manage.py migrate**

- The migrate command looks at the INSTALLED_APPS setting and creates any necessary database tables according to the database settings in your mysite/settings.py file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies. If you're interested, run the command-line client for your database and type \dt (PostgreSQL), SHOW TABLES; (MySQL), or .schema (SQLite) to display the tables Django created.

@ Copyright 2019, Summitworks Technologies Inc.

Understanding QuerySets

- A QuerySet is, in essence, a list of objects of a given Model.
- QuerySets allow you to read the data from the database,

  filter it and order it.
- to define models and to use the database API to create,

  retrieve, update and delete records

```
from django.db import models
from django.utils import timezone


class Post(models.Model):
    author = models.ForeignKey('auth.User',
on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(
            default=timezone.now)
    published_date = models.DateTimeField(
            blank=True, null=True)

    def publish(self):
        self.published_date = timezone.now()
        self.save()

    def __str__(self):
        return self.title
```

@ Copyright 2019, Summitworks Technologies Inc.

QuerySets

- Django shell:

  command-line

  - **(myvenv) ~/djangogirls$ python manage.py shell**

- The effect should be like this:

  command-line

  - (InteractiveConsole)
  - >>>

- You're now in Django's interactive console. It's just like the Python prompt, but with some additional Django magic. You can use all the Python commands here too.

@ Copyright 2019, Summitworks Technologies Inc.

All objects

- Let's try to display all of our posts first. You can do that with the following command:

```
>>> from blog.models import Post
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>]>
```

This is a list of the posts we created earlier! We created these posts using the Django admin interface.

- But now we want to create new posts using Python:**Create object:** This is how you create a new Post object in database:

```
>>> from django.contrib.auth.models import User
>>> me = User.objects.get(username='ola') #'ola' is the superuser we created earlier! Let's get an instance of the user now (adjust this line to use your own username)
>>> me = User.objects.get(username='ola')
>>> Post.objects.create(author=me, title='Sample title', text='Test')
<Post: Sample title>
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>, <Post: Sample title>]>
```

@ Copyright 2019, Summitworks Technologies Inc.

QuerySets: **Filter objects**

- A big part of QuerySets is the ability to filter them. Let's say we want to find all posts that user ola authored. We will use filter instead of all in Post.objects.all(). In parentheses we state what condition(s) a blog post needs to meet to end up in our queryset. In our case, the condition is that author should be equal to me. The way to write it in Django is author=me. Now our piece of code looks like this:

```
>>> Post.objects.filter(author=me)
<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>
```

- If you want to see all the posts that contain the word 'title' in the title field?

```
>>> Post.objects.filter(title__contains='title')
<QuerySet [<Post: Sample title>, <Post: 4th title of post>]>
```

There are two underscore characters (_) between title and contains. Django's ORM uses this rule to separate field names ("title") and operations or filters ("contains"). If you use only one underscore, you'll get an error like "FieldError: Cannot resolve keyword title_contains"

@ Copyright 2019, Summitworks Technologies Inc.

QuerySets

You can also get a list of all published posts. We do this by filtering all the posts that have published_date set in the past:

- **>>> from django.utils import timezone**
- **>>> Post.objects.filter(published_date__lte=timezone.now())**
- **<QuerySet []>**

Unfortunately, the post we added from the Python console is not published yet. But we can change that! First get an instance of a post we want to publish:

command-line

- **>>> post = Post.objects.get(title="Sample title")**

And then publish it with our publish method:

command-line

- **>>> post.publish()**

Now try to get list of published posts again (press the up arrow key three times and hit enter):

command-line

- **>>> Post.objects.filter(published_date__lte=timezone.now())**
- **<QuerySet [<Post: Sample title>]>**

@ Copyright 2019, Summitworks Technologies Inc.

QuerySets: **Ordering objects**

QuerySets also allow you to order the list of objects. Let's try to order them by created_date field:

- **>>> Post.objects.order_by('created_date')**
- **<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>**

We can also reverse the ordering by adding - at the beginning:

- **>>> Post.objects.order_by('-created_date')**
- **<QuerySet [<Post: 4th title of post>, <Post: My 3rd post!>, <Post: Post number 2>, <Post: Sample title>]>**

You can also combine QuerySets by chaining them together:

- **>>> Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')**
- **<QuerySet [<Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>, <Post: Sample title>]>**

@ Copyright 2019, Summitworks Technologies Inc.

Creating Forms from Models

- ModelForm: If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a BlogComment model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.
- For this reason, Django provides a helper class that lets you create a Form class from a Django model.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article
# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']
# Creating a form to add an article.
>>> form = ArticleForm()
# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

@ Copyright 2019, Summitworks Technologies Inc.

Deploy in Windows IIS server

The Django development server is not designed for production use, so using a production-quality web server such as IIS/apache is mandatory for production applications.

Prerequisites for Running Django Applications on IIS:

- Install Python
- Install IIS with CGI
- Create and Configure a Python Virtual Environment for Your Application
- Activate the Virtual Environment and Upgrade pip
- Installed Django and wfastcgi in a Python virtual environment
- Install required packages and Create a Sample Django Application
- Ran the Django project using the Django development server

@ Copyright 2019, Summitworks Technologies Inc.

Deploy in Windows IIS server

The Django development server is not designed for production use, so using a production-quality web server such as IIS/apache is mandatory for production applications.

Prerequisites for Running Django Applications on IIS:

- Install Python
- Install IIS with CGI
- Create and Configure a Python Virtual Environment for Your Application
- Activate the Virtual Environment and Upgrade pip
- Installed Django and wfastcgi in a Python virtual environment
- Install required packages and Create a Sample Django Application
- Ran the Django project using the Django development server

@ Copyright 2019, Summitworks Technologies Inc.

Deploy in Apache Server

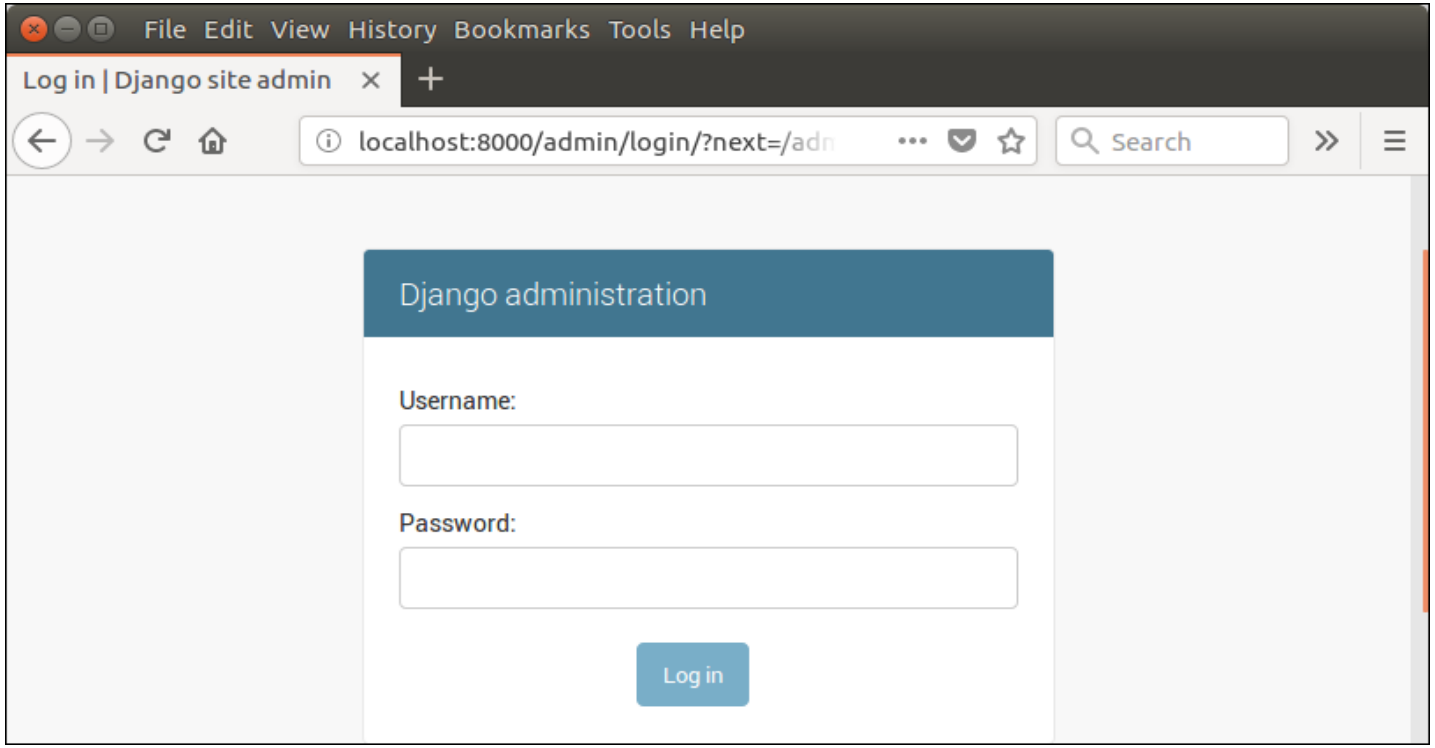
- Prerequisites for Running Django Applications on an Ubuntu server running Apache 2.4.x, using WSGI:

  - Set up a Django site on an Apache virtualhost, using WSGI
  - Use virtualenv to give each site its unique set of Python packages
  - Use WSGI in Daemon mode
  - Run multiple Django sites on one server
  - Deploy new code once we have a site running

@ Copyright 2019, Summitworks Technologies Inc.

Admin Interface

- Django provides a built-in admin module which can be used to perform CRUD operations on the models. It reads metadata from the model to provide a quick interface where the user can manage the content of the application.This is a built-in module and designed to perform admin related tasks to the user.
- The admin app **(django.contrib.admin)** is enabled by default and already added into INSTALLED_APPS section of the settings file.
- To access it at browser use '/**admin**/' at a local machine like **localhost:8000/admin/** and it shows the following output:



@ Copyright 2019, Summitworks Technologies Inc.

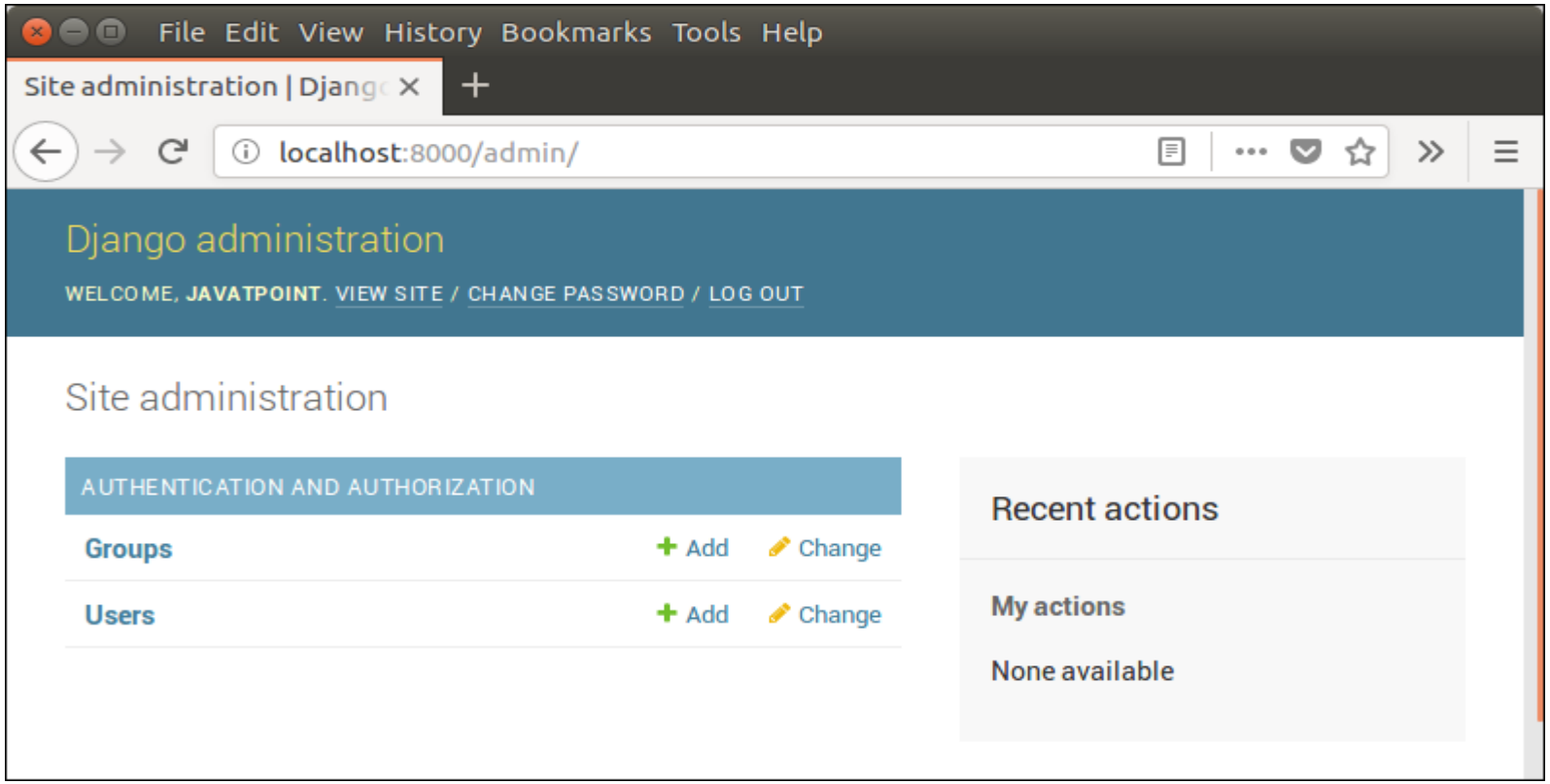
Admin Interface: creating super user

- It prompts for login credentials if no password is created yet, use the following command to create a user.
- Create an Admin User:

```
$ python3 managen.py createsuperuser
```

- After login successfully, it shows the following interface.

It is a Django Admin Dashboard. Here, we can add and update the registered models.



@ Copyright 2019, Summitworks Technologies Inc.

Adding Models to Django Admin

- Let's add our models to the admin site, so we can add, change and delete objects in our custom database tables using this nice admin interface. We'll use three models: Publisher, Author and Book.

admin.py

This code tells the Django admin site to offer an interface for each of these models. Once you've done this, go to your admin home page in your web browser (http://127.0.0.1:8000/admin/), and you should see a "Books" section with links for Authors, Books and Publishers. You might have to stop and start the development server for the changes to take effect.

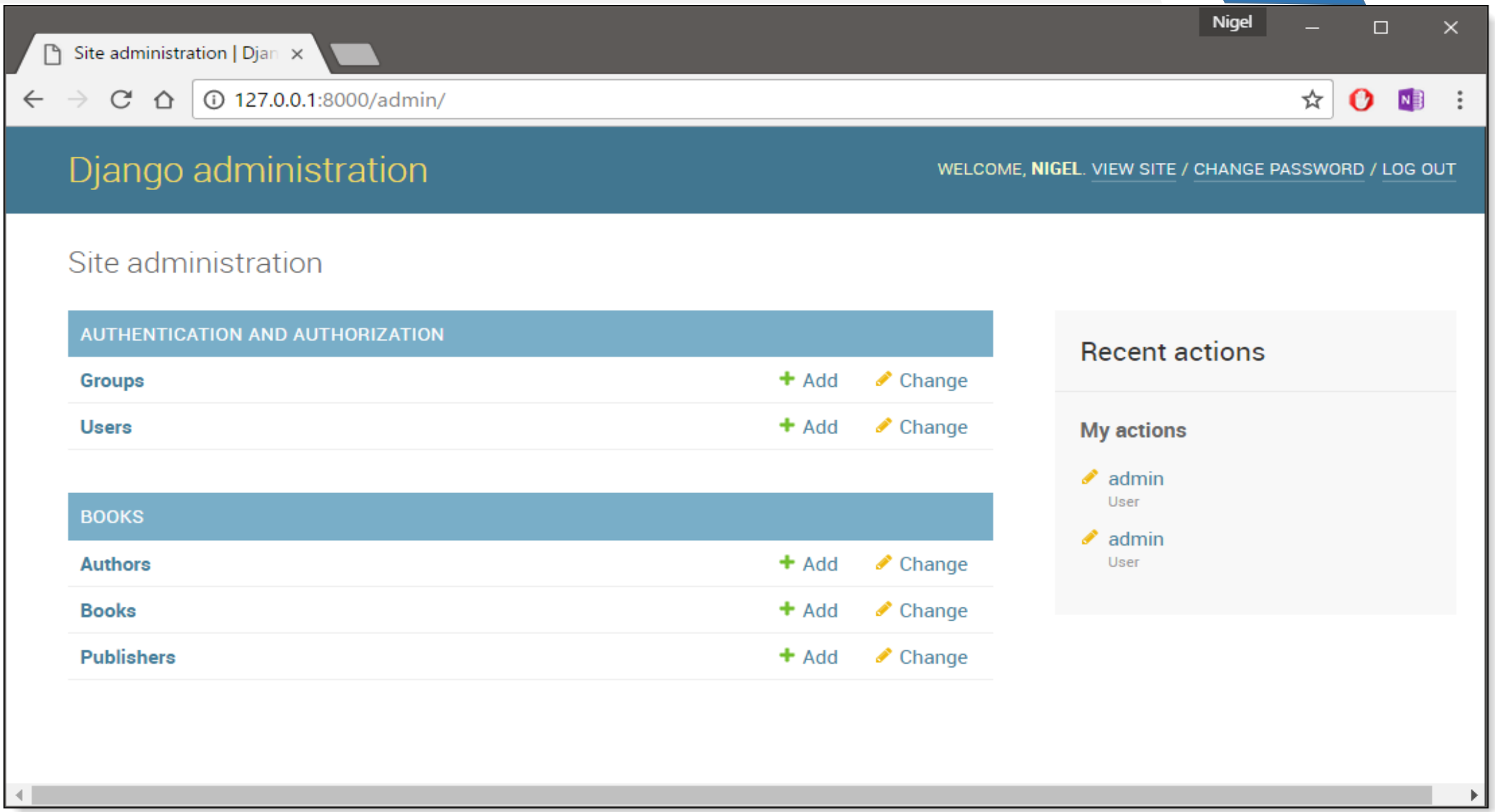
```
from django.contrib import admin
from .models import Publisher, Author, Book

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

@ Copyright 2019, Summitworks Technologies Inc.

Adding Models to Django Admin

You now have a fully functional admin interface for each of those three models. That was easy! Take some time to add and change records, to populate your database with some data.



@ Copyright 2019, Summitworks Technologies Inc.

Customizing Admin interface

- The one thing that I care most to customize is the list view (or Change view rather how the documentation calls it). This is the view that it is used when you select a Model and you get a list with all that Model instances in your DB.
- And to do that what you need to do next is to create a **ModelAdmin** instance.
- the @admin.register() decorator does the same thing as
  - admin.site.register(models.Author, AuthorAdmin).
- what we are doing is registering the **Author** model in the admin app but also telling the app that it needs to use **AuthorAdmin** as **ModelAdmin**.
- The **ModelAdmin** is how you customize its appearance.

```
# admin.py
from django.contrib import admin
import models
@admin.register(models.Author)
class AuthorAdmin(admin.ModelAdmin):
  date_hierarchy = 'created'
  search_fields = ['first_name', 'last_name', 'email']
  list_display = ('first_name','last_name','email')
  list_filter = ('status',)
```

@ Copyright 2019, Summitworks Technologies Inc.

Django Static Files Handling

- In a web application, apart from business logic and data handling, we also need to handle and manage static resources like CSS, JavaScript, images etc.
- It is important to manage these resources so that it does not affect our application performance.
- Django deals with it very efficiently and provides a convenient manner to use resources.
- The **django.contrib.staticfiles** module helps to manage them.

These are the steps to follow to handle static files in Django project

1. Django Static (CSS, JavaScript, images) Configuration: update seetings.py

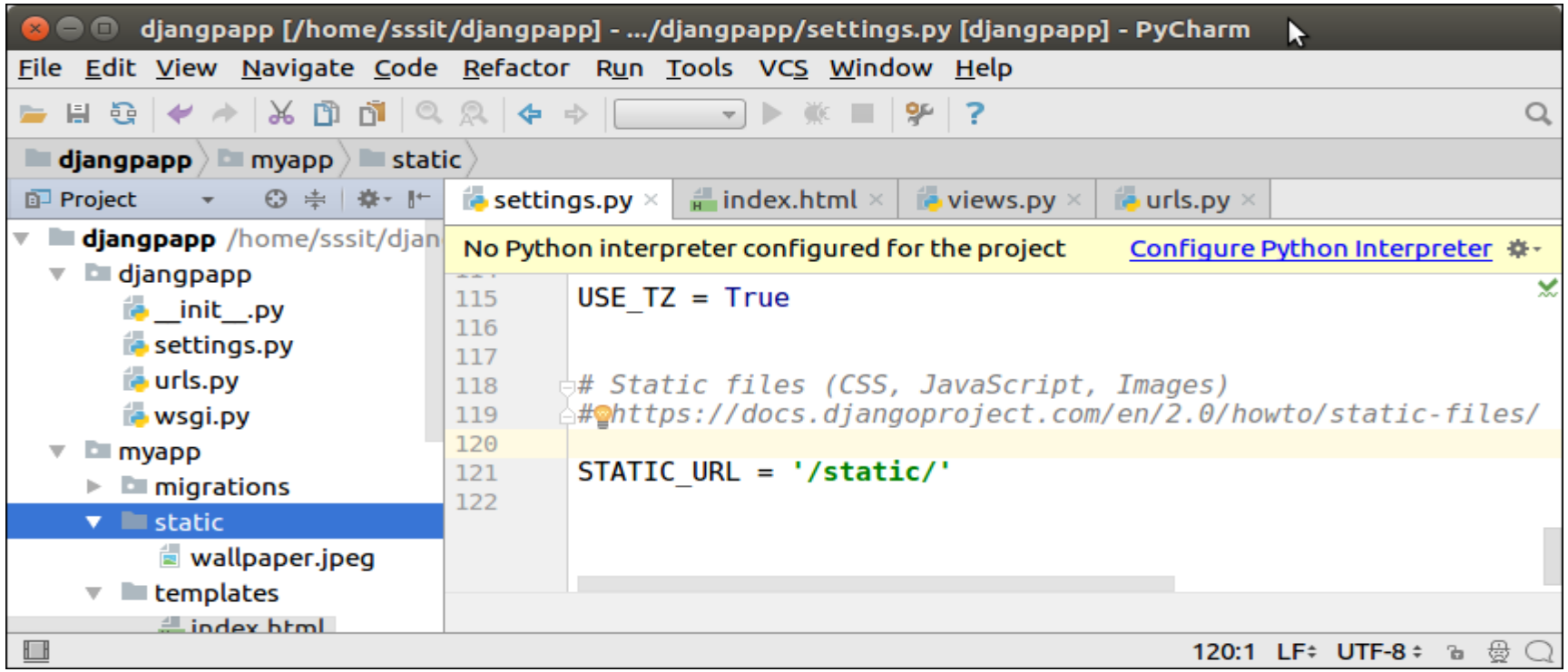
Include the django.contrib.staticfiles in INSTALLED_APPS.

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

@ Copyright 2019, Summitworks Technologies Inc.

Django Static Files Handling: steps

- 2. Define STATIC_URL in settings.py file as given below:
  - STATIC_URL = '/static/'
- 3. Load static files in the templates by using the below expression.
  - {% load **static** %}
- 4. Store all images, JavaScript, CSS files in a **static** folder of the application. First create a directory **static,** store the files inside it.



@ Copyright 2019, Summitworks Technologies Inc.

Django Image Loading Example

- To load an image in a template file, use the code given below.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Index</title>
    {% load static %}
</head>
<body>
<img src="{% static '/wallpaper.jpeg' %}" alt="My image" height="300px"
width="700px"/>
</body>
</html>
```

urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
   path('admin/', admin.site.urls),
   path('index/', views.index),
]
```

views.py

```
def index(request):
   return render(request,'index.html')
```

@ Copyright 2019, Summitworks Technologies Inc.

Django Loading JavaScript

- To load JavaScript file, just add the following line of code in **index.html** file:

{% load static %}

<script src="{% static '/js/script.js' %}"

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Index</title>
    {% load static %}
   <script src="{% static '/js/script.js' %}" type="text/javascript"></script>
</head>
<body>
</body>
</html>
```

script.js

```
alert("Hello, Welcome to Django web app");
```

@ Copyright 2019, Summitworks Technologies Inc.

Django Loading JavaScript: folder structure

djangpapp [/home/sssit/djangpapp] - .../myapp/templates/index.html [djangpapp] - PyCharm

File Edit View Navigate Code Refactor Run Tools VCS Window Help

djangpapp > myapp > static > js > script.js

Project

- settings.py
- urls.py
- wsgi.py
- myapp
  - migrations
  - static
    - js
      - script.js
    - wallpaper.jpeg
  - templates
    - index.html

settings.py | index.html | views.py | urls.py

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Index</title>
6       {% load static %}
7      <script src="{% static '/js/script.js' %}" type="t
8
9  </head>
10 <body>
11
12 </body>
```

html › body

227 chars, 12 line breaks 11:1 LF UTF-8

@ Copyright 2019, Summitworks Technologies Inc.

Django Loading CSS Example

- To, load CSS file, use the following code in **index.html** file.

```
{% load static %}
<link href="{% static 'css/style.css' %}" rel="stylesheet">
```

- After that create a directory CSS and file style.css which contains the following code.

style.css

```
h1{
color:red;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Index</title>
    {% load static %}
   <link href="{% static 'css/style.css' %}" rel="stylesheet">
</head>
<body>
<h1>Welcome to Javatpoint</h1>
</body>
</html>
```

@ Copyright 2019, Summitworks Technologies Inc.

Django Loading CSS Example: folder structure

djangpapp [/home/sssit/djangpapp] - .../myapp/templates/index.html [djangpapp] - PyCharm

File Edit View Navigate Code Refactor Run Tools VCS Window Help

djangpapp > myapp > static > css > style.css

Project

- settings.py
- urls.py
- wsgi.py
- myapp
  - migrations
  - static
    - css
      - style.css
    - js
      - script.js
    - wallpaper.jpeg

settings.py | index.html | views.py | urls.py

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Index</title>
6       {% load static %}
7      <link href="{% static 'css/style.css' %}" rel="sty
8
9  </head>
10 <body>
11 <h1>Welcome to Javatpoint</h1>
12 </body>
```

html › head

8:1 LF UTF-8

@ Copyright 2019, Summitworks Technologies Inc.

User authentication

- The Django authentication system handles both authentication and authorization. Briefly, authentication verifies a user is who they claim to be, and authorization determines what an authenticated user is allowed to do.
- Django's authentication system in its default configuration has evolved to serve the most common project needs, handling a reasonably wide range of tasks, and has a careful implementation of passwords and permissions.
- The Django auth app: Django automatically installs the auth app when a new project is created. Look in the settings.py under INSTALLED_APPS

  and you can see auth is one of several built-in apps

  Django has installed for us.

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

@ Copyright 2019, Summitworks Technologies Inc.

User authentication

- To use the auth app we need to add it to our project-level urls.py file.

```
# my_project/urls.py
from django.contrib import admin
 from django.urls import path, include
urlpatterns = [
path('admin/', admin.site.urls),
path('accounts/', include('django.contrib.auth.urls')),
]
```

- On the second line we're also importing include from django.urls. And I've chosen to include the auth app at accounts/ but you can use any url pattern you want.

- The auth app we've now included provides us with several authentication views and URLs for handling login, logout, and password management.

@ Copyright 2019, Summitworks Technologies Inc.

Internationalization

- Refers to the process of designing programs for the potential use of any locale. This process is usually done by software developers. Internationalization includes marking text (such as UI elements and error messages) for future translation, abstracting the display of dates and times so that different local standards may be observed, providing support for differing time zones, and generally making sure that the code contains no assumptions about the location of its users. You'll often see internationalization abbreviated *I18N*. (The 18 refers to the number of letters omitted between the initial I and the terminal N.)

Steps are:

1. First, we will create a folder to save all the translation files.
2. Next, open your settings/base.py file and make sure you have, USE_I18N = True
3. and the template context processor django.template.context_processors.i18n is inside the TEMPLATES['OPTIONS']['context_processors'] setting:

```
TEMPLATES = [
   {
'OPTIONS': {
          'context_processors': [
              ...
              'django.template.context_processors.i18n',
          ],
       },   },
]
```

@ Copyright 2019, Summitworks Technologies Inc.

Internationalization

- add the Locale middleware in the correct position, to be able to determine the user's language preferences through the request context:

```
MIDDLEWARE_CLASSES = (
    ...

'django.contrib.sessions.middleware.SessionMiddle
ware',
    'django.middleware.locale.LocaleMiddleware',

'django.middleware.common.CommonMiddleware',
    ...
)
```

- Next, specify the languages you want to use:

```
from django.utils.translation import ugettext_lazy as _
LANGUAGES = (
    ('en', _('English')),
    ('ca', _('Catalan')),
)
```

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework

- To cache something is to save the result of an expensive calculation, so that you don't perform it the next time you need it. Following is a pseudo code that explains how caching works.

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

- Django comes with its own caching system that lets you save your dynamic pages, to avoid calculating them again when needed. The good point in Django Cache framework is that you can cache −
    - The output of a specific view.
    - A part of a template.
    - Your entire site.
- To use cache in Django, first thing to do is to set up where the cache will stay. The cache framework offers different possibilities - cache can be saved in database, on file system or directly in memory. Setting is done in the **settings.py** file of your project.

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework: Memcached

- The fastest, most efficient type of cache supported natively by Django, Memcached is an entirely memory-based cache server, originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It is used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.
- Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework:Database caching

- Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.
- To use a database table as your cache backend:
  - Set BACKEND to django.core.cache.backends.db.DatabaseCache
  - Set LOCATION to tablename, the name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.
- In this example, the cache table's name is my_cache_table:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django’s Cache Framework: Filesystem caching

- the file-based backend serializes and stores each cache value as a separate file. To use this backend set BACKEND to "django.core.cache.backends.filebased.FileBasedCache" and LOCATION to a suitable directory. For example, to store cached data in /var/tmp/django_cache, use this setting:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.filebased.FileBasedCache'
,
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

If you’re on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework: Local-memory caching

- This is the default cache if another is not specified in your settings file. If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is per-process (see below) and thread-safe. To use it, set BACKEND to

  "django.core.cache.backends.locmem.LocMemCache".

  For example:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework: Dummy caching (for development)

- Django comes with a "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.
- This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set BACKEND like so:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.dummy.DummyCache',
    }
}
```

@ Copyright 2019, Summitworks Technologies Inc.

Django's Cache Framework: Template fragment caching

- If you're after even more control, you can also cache template fragments using the cache template tag. To give your template access to this tag, put {% load cache %} near the top of your template.

- The {% cache %} template tag caches the contents of the block for a given amount of time. It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. The fragment is cached forever if timeout is None. The name will be taken as is, do not use a variable. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

@ Copyright 2019, Summitworks Technologies Inc.

Any Queries

@ Copyright 2019, Summitworks Technologies Inc.