Everything You Should Know About Django

Introduction

Django is a free and open-source, Python-based web framework that follows the model-template-views (MTV) architectural pattern. It is maintained by the Django Software Foundation (DSF), an independent organization established in the US as a non-profit.

Django's primary goal is to ease the creation of complex, database-driven websites. The framework emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don't repeat yourself.Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models

Some well-known sites that use Django include:

- Instagram
- Mozilla
- Bitbucket
- Nextdoor

What you should already know

Despite having its own nomenclature, such as naming the callable objects generating the HTTP responses "views", the core Django framework can be seen as an MVC architecture. It consists of an object-relational mapper (ORM) that mediates between data models (defined as Python classes) and a relational database ("Model"), a system for processing HTTP requests with a web templating system ("View"), and a regular-expression-based URL dispatcher ("Controller").

Also included in the core framework are:

Everything You Should Know About Django

- a lightweight and standalone web server for development and testing
- a form serialization and validation system that can translate between HTML forms and values suitable for storage in the database
- an internationalization system, including translations of Django's own components into a variety of languages
- a system for extending the capabilities of the template engine
- an interface to Python's built-in unit test framework

Hello world

To begin, open up a new command line shell or use the built-in terminal on VS Code. For the latter click on “Terminal” at the top and then “New Terminal” to bring it up on the bottom of the screen.

Make sure you are not in an existing virtual environment by checking there is nothing in parentheses before your command line prompt. You can even type deactivate to be completely sure. Then navigate to the code directory on your Desktop and create a helloworld directory with the following commands.

```
# Windows
> cd onedrive\desktop\code
> mkdir helloworld
> cd helloworld

# macOS
% cd ~/desktop/code
% mkdir helloworld
% cd helloworld
```

Create a new virtual environment called .venv, activate it, and install Django with Pip

Everything You Should Know About Django


```
# Windows
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django~=4.0.0

# macOS
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django~=4.0.0
```

Models

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; see Making queries.

Using Models

Once you have defined your models, you need to tell Django you're going to use those models. Do this by editing your settings file and changing the INSTALLED_APPS setting to add the name of the module that contains your models.py.

For example, if the models for your application live in the module myapp.models (the package structure that is

created for an application by the manage.py startapp script), INSTALLED_APPS should read, in part:

```
INSTALLED_APPS = [
#...
'myapp',
#...
]
```

When you add new apps to INSTALLED_APPS, be sure to run manage.py migrate, optionally making migrations for them first with manage.py makemigrations.

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.

The above Person model would create a database table like this:

```
CREATE TABLE myapp_person (
"id" serial NOT NULL PRIMARY KEY,
"first_name" varchar(30) NOT NULL,
"last_name" varchar(30) NOT NULL
);
```

Fields

The most important part of a model - and the only required part of a model - is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like clean, save, or delete.

Example:

```
from django.db import models

class Musician(models.Model):
first_name = models.CharField(max_length=50)
last_name = models.CharField(max_length=50)
instrument = models.CharField(max_length=100)

class Album(models.Model):
artist = models.ForeignKey(Musician,
on_delete=models.CASCADE)
name = models.CharField(max_length=100)
release_date = models.DateField()
num_stars = models.IntegerField()
```

Field Types

Each field in your model should be an instance of the appropriate Field class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. INTEGER, VARCHAR, TEXT).
- The default HTML widget to use when rendering a form field. The minimal validation requirements, used in Django's admin and in automatically-generated forms.
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Django ships with dozens of built-in field types; you can find the complete list in the model field reference. You can easily write your own fields if Django's built-in ones don't do the trick; see How to create custom model fields.

Field Options

Each field takes a certain set of field-specific arguments (documented in the model field reference). For example, CharField (and its subclasses) require a max_length argument which specifies the size of the VARCHAR database field used to store the data.

There's also a set of common arguments available to all field types. All are optional. The're fully explained in the reference, but here's a quick summary of the most often-used ones:

null
If True, Django will store empty values as NULL in the database. Default is False.

blank
If True, the field is allowed to be blank. Default is False.

Note that this is different than null. null is purely database-related, whereas blank is validation-related. If a field has blank=True, form validation will allow entry of an empty value. If a field has blank=False, the field will be required.

choices
A sequence of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = [
('FR', 'Freshman'),
```

```
('SO', 'Sophomore'),
('JR', 'Junior'),
('SR', 'Senior'),
('GR', 'Graduate'),
]
```

The first element in each tuple is the value that will be stored in the database. The second element is displayed by the field's form widget.

Given a model instance, the display value for a field with choices can be accessed using the get_FOO_display() method. For example:

```
from django.db import models

class Person(models.Model):
SHIRT_SIZES = (
('S', 'Small'),
('M', 'Medium'),
('L', 'Large'),
)
name = models.CharField(max_length=60)
shirt_size = models.CharField(max_length=1,
choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone",
shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

You can also use enumeration classes to define choices in a concise way:

```
from django.db import models
```

```
class Runner(models.Model):
MedalType = models.TextChoices('MedalType',
'GOLD SILVER BRONZE')
name = models.CharField(max_length=60)
medal = models.CharField(blank=True,
choices=MedalType.choices, max_length=10)
```

Making Queries

Once you've created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the data model reference for full details of all the various model lookup options.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a blog application:

```
from datetime import date

from django.db import models

class Blog(models.Model):
name = models.CharField(max_length=100)
tagline = models.TextField()

def __str__(self):
return self.name

class Author(models.Model):
name = models.CharField(max_length=200)
email = models.EmailField()

def __str__(self):
return self.name

class Entry(models.Model):
```

```
blog = models.ForeignKey(Blog,
on_delete=models.CASCADE)
headline = models.CharField(max_length=255)
body_text = models.TextField()
pub_date = models.DateField()
mod_date = models.DateField(default=date.today)
authors = models.ManyToManyField(Author)
number_of_comments =
models.IntegerField(default=0)
number_of_pingbacks =
models.IntegerField(default=0)
rating = models.IntegerField(default=5)

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

Performing Raw Queries

The raw() manager method can be used to perform raw SQL queries that return model instances:

Manager.raw(raw_query, params=(), translations=None)

This method takes a raw SQL query, executes it, and returns a django.db.models.query.RawQuerySet instance. This RawQuerySet instance can be iterated over like a normal QuerySet to provide object instances.

This is best illustrated with an example. Suppose you have the following model:

```
class Person(models.Model):
first_name = models.CharField(...)
last_name = models.CharField(...)
birth_date = models.DateField(...)
```

Lookup API Reference

This document has the API references of lookups, the Django API for building the WHERE clause of a database query. To learn how to use lookups, see Making queries; to learn how to create new lookups, see How to write custom lookups.

The lookup API has two components: a RegisterLookupMixin class that registers lookups, and the Query Expression API, a set of methods that a class has to implement to be registrable as a lookup.
Django has two base classes that follow the query expression API and from where all Django builtin lookups are derived:

- Lookup: to lookup a field (e.g. the exact of field_name__exact)
- Transform: to transform a field

A lookup expression consists of three parts:

- Fields part (e.g. Book.objects.filter(author__best_friends__first_name...);
- Transforms part (may be omitted) (e.g. __lower__first3chars__reversed);
- A lookup (e.g. __icontains) that, if omitted, defaults to __exact.

Reference

- All the documentation in this page is taken from [Django Organisation]