

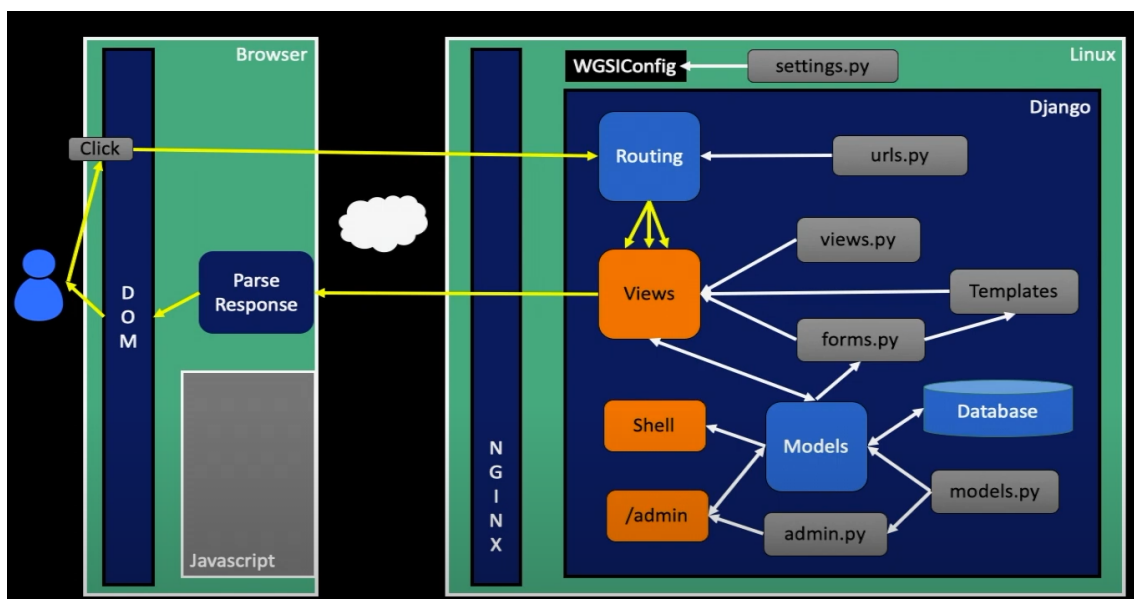


Django - Python

▼ Django

▼ MVT

- Modelo: Donde creas las clases que interactúan con la base de datos.
- “Vista”: La manera en la que se gestiona lo que vas a ver, interactúa con la database a través de ORM. Es vista porque es la lógica de cómo se te va a presentar lo que ves (según el equipo de Django). Junto a las urls sería el controlador en el MVC.
- Template: La vista del MVC. Literalmente el html que ve la gente.



▼ Estructura

Por cada conjunto de casos de uso se tiene una aplicación (el núcleo y otras apps)

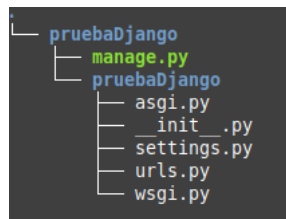
└─ pruebaDjango (proyecto)

```

└─
manage.py (comandos que te ayudan)
└─ pruebaDjango (núcleo)

└─
asgi.py (despliegue producción)
└─
init.py
└─
settings.py (configs, datos guardados)
└─
urls.py (expresiones regulares)
└─
wsgi.py (despliegue producción)
admin.py CRUD para usuarios-admins
apps.py Elementos config
migration migraciones base de datos
models.py
tests.py tests

```



▼ Starting commands

▼ Virtual environments:

Un espacio virtual, pensado para poder instalar dependencias en tu ordenador o MV, separadas de otras (imagina que en un proyecto trabajas con Python 3.1 y en otro Python 2.algo).

<https://virtualenvwrapper.readthedocs.io/en/latest/#>

▼ Primeros pasos

```

$ pip install virtualenvwrapper
...
$ export WORKON_HOME=~/.Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv env1

```

1. El virtualenvwrapper facilita cambiar entre entornos virtuales, crearlos etc
2. Decides en qué carpeta se van a guardar los ev. en este caso Envs.

▼ Explanation detailed

Sure, let me explain that command in a simple way!

```
export WORKON_HOME=~/.Envs
```

Imagine you have a special folder where you want to keep all your toy boxes. This command is like telling your friends, "Hey, my special folder for toy boxes is called 'Envs'".

Now, let's break it down:

- `export`: It's like saying "I want everyone to know about this."
- `WORKON_HOME`: This is like a nickname for your special folder.
- `~/Envs`: This is the path or address of your special folder. The `~` means "my home" (like your house). So, it says, "My special folder is in my home, and it's called 'Envs'."

Putting it all together, the command says, "I want everyone to know that my special folder for toy boxes is in my home, and it's called 'Envs'." This special folder is where we'll keep our Python environments (kind of like special places for different types of toys).

3. Creas el directorio en caso de que no existiera (-p)

4. `virtualenvwrapper.sh` script le dices dónde está para que funcione.

5. set virtualenvproject en el directorio que quieras que te abra x defecto

Si te da error, lo más seguro es que se haya instalado en otra ruta en ese caso

▼ Error `virtualenvrappwer.sh` not found

Comprueba que esté instalado:

```
pip show virtualenvwrapper
```

Y luego si todo ok lo buscas así:

```
find / -name virtualenvwrapper.sh 2>/dev/null
```

- `find`: This is the command to search for files.
- `/`: This is the starting point of the search. It means "start searching from the root directory."
- `name virtualenvwrapper.sh`: This part tells `find` to look for a file with the name `virtualenvwrapper.sh`.
- `2>/dev/null`: This is a way to suppress error messages that might occur if the command doesn't have permission to access certain directories.

Now, let's say your username is `your_username`. You might modify the command like this:

bashCopy code

```
find /home/your_username -name virtualenvwrapper.sh 2>/dev/null
```

Finalmente, vuelves a hacer el source, pero con la ruta bien puesta.

Rayada máxima si no te pillas el comando pese a estar instalado:

maybe path issues?

https://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html#path-management

```
minty@minty-VB:~/local/bin$ ls
markdown-it  pycheck      virtualenv    virtualenvwrapper_lazy.sh
pbr          pygmentize   virtualenv-clone  virtualenvwrapper.sh
minty@minty-VB:~/local/bin$ workon env1
workon: command not found
minty@minty-VB:~/local/bin$ source virtualenvwrapper.sh
minty@minty-VB:~/local/bin$ workon env1
(env1) minty@minty-VB:~/local/bin$
```

.bashrc en el home, última línea source y le das la ruta que es.

```

virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/premkproject
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/postmkproject
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/initialize
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/premkvirtualenv
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/postmkvirtualenv
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/prermvirtualenv
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/postrmvirtualenv
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/predeactivate
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/postdeactivate
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/preactivate
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/postactivate
virtualenvwrapper.user_scripts creating /home/minty/.virtualenvs/get_env_details

```

▼ Crear y cambiar de environment

- mkvirtualenv para crearlos
- workon para cambiar entre envs

```

mkvirtualenv ejemplo1
workon ejemplo2

```

▼ Install and create project

▼ Iniciando

- [Step 1 | Install Django](#)
- [Step 2 | Start Project](#)
- [Step 3 | Create app](#)
- [Step 4 | Add app to settings.py](#)
- mkdir y cd → `django-admin startproject nombreBase`
- cd nombre (carpeta raíz)→ `python manage.py startapp nombreApp`
- en settings de tu nombreBase añades a installed apps:

```

INSTALLED_APPS = [
    'recipes.apps.RecipesConfig' ,

```

▼ Templates

- Creas la carpeta templates dentro de la app y otra carpeta con el nombre de la app:

appname (store) --> templates --> appname (store)

(esto es para que no haya conflicto y puedes tener dos archivos que se llamen igual, pero la ruta sea diferente según la app a la que pertenezcan)

- creas templates: ejemplos de código básico:

▼ Views

- ejemplo código con import render y def:

```

from django.shortcuts import render

def store(request):
    context = {}
    return render(request, 'store/store.html', context)

def cart(request):
    context = {}
    return render(request, 'store/cart.html', context)

def checkout(request):
    context = {}
    return render(request, 'store/checkout.html', context)

```

- ejemplo con class Generic views

▼ Urls

▼ Link routes core urls.py

En nombreBase urls.py le añades el include al import de django.urls y le añades los archivos urls de tus apps:

```
///File: nombreBase/urls.py
from django.contrib import admin
from django.urls import path, include

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

- Basic routes:

```
///File: store/urls.py

from django.urls import path
from . import views

app_name = "store.py"
urlpatterns = [
    #Leave as empty string for base url
    path('', views.store, name="store"),
    path('cart/', views.cart, name="cart"),
    path('checkout/', views.checkout, name="checkout"),
]
```

- Generic views y media:

▼ Models

▼ admin.py

En tu app donde hayas hechos los models tienes que importar en admining.py:

```
from django.contrib import admin

# Register your models here.
from .models import *

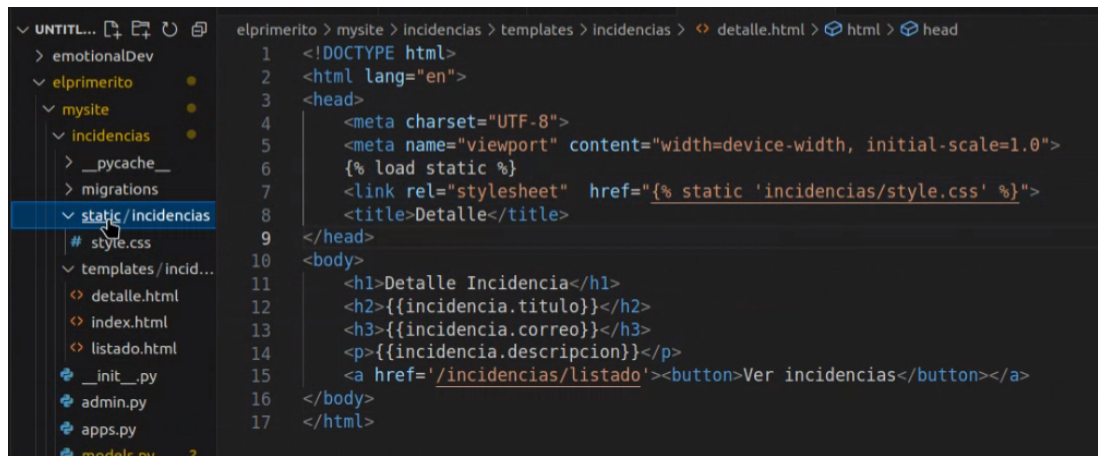
admin.site.register(User)
```

▼ .gitignore

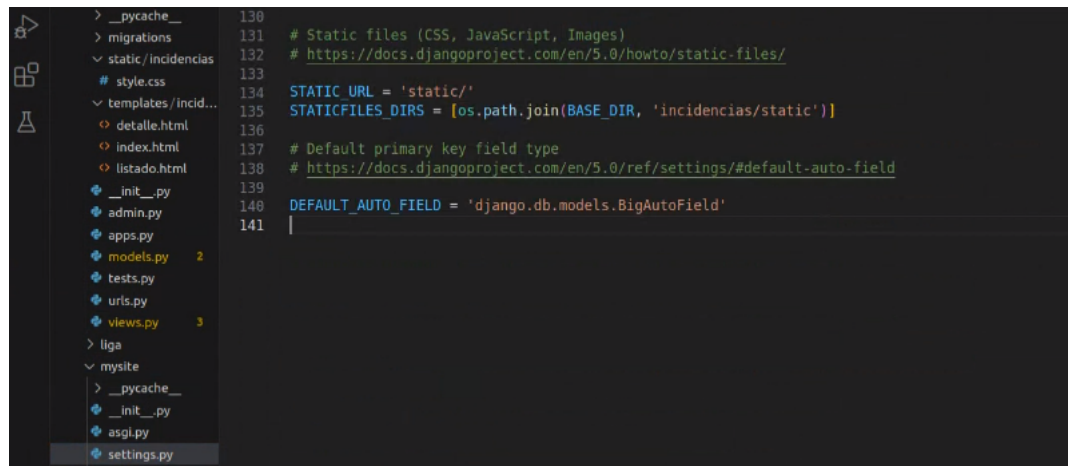
- gitignore.io → web para todo tipo de proyectos
- <https://djangowaves.com/tips-tricks/gitignore-for-a-django-project/> (mejor explicación pero es de 2020)
- <https://intellij-support.jetbrains.com/hc/en-us/articles/206544839-How-to-manage-projects-under-Version-Control-Systems>. (Pycharm)

▼ static css etc

también hay un paquete de django bootstrap



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   {% load static %}
7   <link rel="stylesheet" href="{% static 'incidencias/style.css' %}">
8   <title>Detalle</title>
9 </head>
10 <body>
11   <h1>Detalle Incidencia</h1>
12   <h2>{{ incidencia.titulo }}</h2>
13   <h3>{{ incidencia.correo }}</h3>
14   <p>{{ incidencia.descripcion }}</p>
15   <a href="/incidencias/listado"><button>Ver incidencias</button></a>
16 </body>
17 </html>
```



```
130
131
132 # Static files (CSS, JavaScript, Images)
133 # https://docs.djangoproject.com/en/5.0/howto/static-files/
134
135 STATIC_URL = 'static/'
136 STATICFILES_DIRS = [os.path.join(BASE_DIR, 'incidencias/static')]
137
138 # Default primary key field type
139 # https://docs.djangoproject.com/en/5.0/ref/settings/#default-auto-field
140
141 DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

▼ Database

▼ Modifications and migrations

- Change your models (in `models.py`).
 - Run `python manage.py makemigrations` to create migrations for those changes
 - Run `python manage.py migrate` to apply those changes to the database.
- `python manage.py check`; this checks for any problems in your project without making migrations or touching the database.

▼ How migrate works

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.

▼ Atributos

▼ Texto

CharField y TextField son ambos campos de texto en Django, pero se utilizan en diferentes contextos:

CharField se utiliza para almacenar cadenas de caracteres de longitud limitada. Debes especificar el argumento `max_length` que define la longitud máxima de los caracteres que se pueden guardar en este campo. Es ideal para almacenar datos de texto cortos como nombres, títulos, etc.

TextField se utiliza para almacenar grandes cantidades de texto. No necesitas especificar una longitud máxima para este campo. Es ideal para almacenar datos de texto largos como descripciones, contenido de artículos, etc.

```
class MyModel(models.Model):
    short_text = models.CharField(max_length=200) # Para texto corto
    long_text = models.TextField() # Para texto largo
```

▼ Imágenes

Para poder subir una imagen como atributo en una clase en Django, necesitas seguir los siguientes pasos:

Asegúrate de tener instalada la librería Pillow. Esta librería se utiliza para trabajar con imágenes en Django. Puedes instalarla con pip:

```
pip install pillow
```

En tu modelo, define un campo ImageField. Ya lo has hecho en tu modelo Juego: El argumento upload_to define el subdirectorío en el que se guardarán las imágenes cargadas.

```
image = models.ImageField(upload_to='juego_images/')
```

En tu configuración de Django (normalmente en el archivo `settings.py`), asegúrate de tener definida la configuración MEDIA_ROOT y MEDIA_URL. MEDIA_ROOT es el directorio absoluto donde se guardarán los archivos cargados, y MEDIA_URL es la URL que se utilizará para referenciar estos archivos:

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

En tu archivo

`urls.py` principal, añade una ruta para servir los archivos media en modo desarrollo:

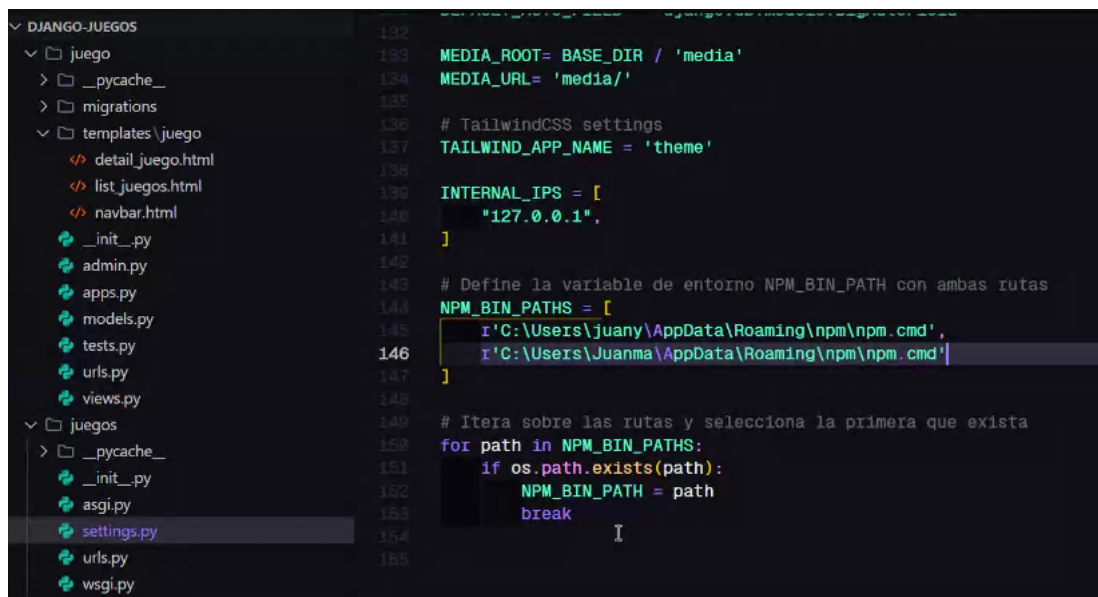
```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... tus otras rutas aquí ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

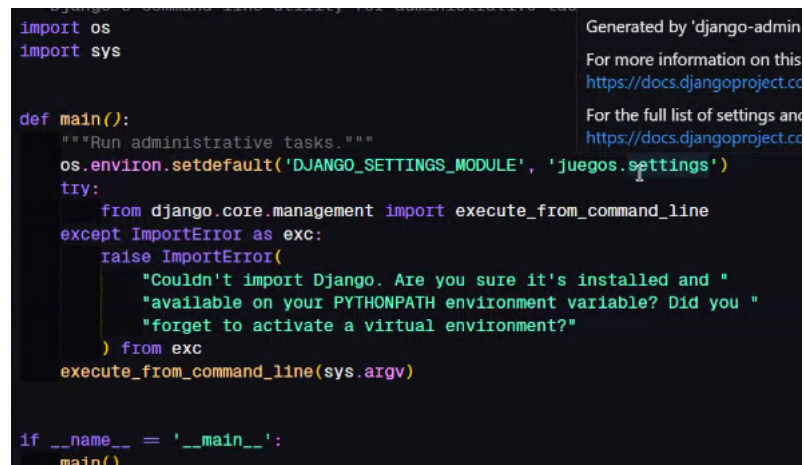
Nota: Servir archivos media directamente con Django solo se recomienda en modo desarrollo. En un entorno de producción, deberías utilizar un servidor web como Nginx o un servicio de almacenamiento como Amazon S3 para servir tus archivos estáticos y media.

Finalmente, en tu formulario de Django o en tu API, deberías ser capaz de aceptar un archivo de imagen y guardarlo en tu modelo Juego.

▼ ENVIS



O así, o una variable que según el os cargue aquí un settings u otro:



▼ Users y herencia

Caso de users donde hay una clase predefinida y tú quieres extenderla: Opción 1:
modelo 1:1 user:profile

```

from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    # Añade aquí cualquier campo adicional que necesites. Por ejemplo:
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=30, blank=True)
    birth_date = models.DateField(null=True, blank=True)

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)

```



```
@receiver(post_save, sender=User)
def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()
```

Opción 2: extender y modificar la clase user

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

class User(AbstractUser):
    # Agregar campos personalizados si es necesario
    bio = models.TextField(blank=True)
    profile_picture = models.ImageField(upload_to='profile_pictures/', blank=True)
    favourite_recipes = models.ManyToManyField('recipes.Recipe', related_name='fa

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

Tu enfoque (modelo Profile con OneToOneField a User):

Pros:

- Mantienes la separación entre los datos de autenticación del usuario (User) y los datos del perfil del usuario (Profile). Esto puede ser útil si los datos del perfil son opcionales o si solo se necesitan en ciertas partes de tu aplicación.
- Puedes añadir o cambiar campos en el modelo Profile sin afectar al modelo User.

Contras:

- Necesitas manejar la creación y el guardado del perfil de usuario en las señales de post_save.
- Para acceder a los datos del perfil, necesitas hacer una consulta adicional a la base de datos (por ejemplo, `user.profile.bio`).

Enfoque alternativo (extender AbstractUser):

Pros:

- Todos los datos del usuario están en un solo lugar, lo que puede simplificar las consultas a la base de datos.
- No necesitas manejar las señales de post_save para crear o guardar el perfil del usuario.

Contras:

- Estás modificando el modelo User, lo que puede tener implicaciones si Django cambia este modelo en futuras versiones.
- Si tienes muchos campos que solo son relevantes para ciertas partes de tu aplicación, estos campos estarán presentes en todas las instancias de User, lo que puede consumir más memoria.

En resumen, si los datos del perfil son una parte integral de tu usuario y siempre los necesitarás cuando manejes usuarios, podría tener sentido extender AbstractUser. Si los datos del perfil son opcionales o solo los necesitas en ciertas partes de tu aplicación, podría ser mejor mantenerlos en un modelo separado.

Opción 3:

Sí, puedes crear una clase que herede de `User` o `AbstractUser` sin modificar las clases base. Esto se llama "extensión de clase" y es una característica común de la programación orientada a objetos. Aquí te dejo un ejemplo de cómo podrías hacerlo:

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

```

from django.db import models

class CustomUser(AbstractUser):
    # Agregar campos personalizados si es necesario
    bio = models.TextField(blank=True)
    profile_picture = models.ImageField(upload_to='profile_pictures/', blank=True)

    def __str__(self):
    return self.username

```

En este código, `CustomUser` es una nueva clase que hereda de `AbstractUser`. Esto significa que `CustomUser` tiene todos los campos y métodos de `AbstractUser`, además de los campos adicionales que definimos (`bio` y `profile_picture`).

Para usar `CustomUser` en lugar de `User` para la autenticación, necesitarás hacer algunos cambios adicionales en tu configuración de Django. En particular, necesitarás establecer la opción `AUTH_USER_MODEL` en tu archivo de configuración `settings.py` para apuntar a tu nuevo modelo de usuario:

```
AUTH_USER_MODEL = 'myapp.CustomUser'
```

Reemplaza `'myapp'` con el nombre de la aplicación donde se encuentra tu modelo `CustomUser`.

▼ Templates

▼ Models

▼ Fields

Opcionales:

```

from django.db import models

class Recipe(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    ingredients = models.TextField()
    steps = models.TextField()
    tags = models.CharField(max_length=200, blank=True, null=True)
    image = models.ImageField(upload_to='recipes/', blank=True, null=True)

```

En este código, `blank=True` permite que el campo sea vacío en los formularios, y `null=True` permite que el campo sea `NULL` en la base de datos. Los campos sin estas opciones serán obligatorios.

Si ya tienes datos en tu base de datos y cambias un campo de opcional a obligatorio, tendrás que proporcionar un valor por defecto o una migración de datos para llenar los valores existentes, o Django te dará un error cuando intentes hacer la migración.

Para dar un valor por defecto a un campo en un modelo de Django, puedes usar el argumento `default` en la definición del campo. Por ejemplo, si tienes un campo `title` y quieres que el valor por defecto sea `"Untitled"`, puedes hacerlo así:

```
title = models.CharField(max_length=200, default="Untitled")
```

Para la validación, Django realiza algunas automáticamente, como verificar que los campos obligatorios no estén vacíos y que los datos ingresados sean del tipo correcto. Si quieres agregar validación adicional, puedes hacerlo en el método `clean` del modelo. Por ejemplo, si quieres validar que el `title` tenga al menos 5 caracteres, puedes hacerlo así:

```

from django.core.exceptions import ValidationError

class Recipe(models.Model):
    title = models.CharField(max_length=200, default="Untitled")

```

```
# other fields...

def clean(self):
    if len(self.title) < 5:
        raise ValidationError("Title must be at least 5 characters long.")
```

En este código, `ValidationError` es una excepción que puedes lanzar para indicar que los datos no son válidos. Cuando llamas a `model.full_clean()`, Django llamará al método `clean` y agregará cualquier error que encuentre a los errores del modelo.

Recuerda que la validación del modelo no se ejecuta automáticamente cuando guardas un objeto. Tienes que llamar a `full_clean()` explícitamente si quieres que se ejecute la validación. En un `ModelForm`, la validación del modelo se ejecuta automáticamente cuando llamas a `form.is_valid()`.

▼ Pasos

Aparte de crear los modelos, hacer urls y views que los manejen, en `tuapp/admin.py` tienes que “registrarlos”:

```
from django.contrib import admin
from .models import Question
admin.site.register(Question)
```

▼ Generic Views

Parece magia, las más útiles para mí `ListView`, `DetailView`, y las CRUD (`CreateView`, etc). La documentación de Django no lo explica mucho, pero este tipo sí:

<https://www.youtube.com/watch?v=uPRabryhv5k&t=211s>

▼ `get_object_or_404`

A shortcut: `get_object_or_404()`.

It’s a very common idiom to use `get()` and raise `Http404` if the object doesn’t exist. Django provides a shortcut. Here’s the `detail()` view, rewritten:

```
polls/views.py
from django.shortcuts import get_object_or_404, render

from .models import Question

# ... def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, "polls/detail.html", {"question": question})
```

The `get_object_or_404()` function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the `get()` function of the model’s manager. It raises `Http404` if the object doesn’t exist.

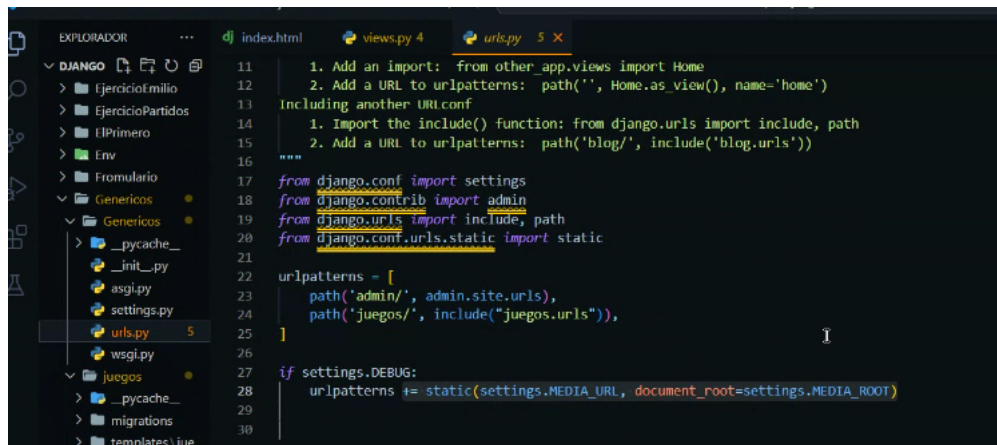
There’s also a `get_list_or_404()` function, which works just as `get_object_or_404()` – except using `filter()` instead of `get()`. It raises `Http404` if the list is empty.

▼ Url static

En producción no deberíamos tener esto porque el server ya se carga de servir los archivos estáticos. Así que

En dev los static se sirven cambiando la config y media no, hay que instalar pillow y añadirlo a ruta o...

Y al pasar a Prod hay que en settings general hay que poner el allowed host al host que sea, y el debug a false. Entonces entiende que estás en prod y deja de servirlos.



If you want to dynamically generate the URL for an image based on the `recipe.image` field, you can do so directly in your Django template. Here's an example:

```

```

This will automatically generate the correct URL for the image, based on your Django storage configuration.

▼ Allauth

Cuando utilizas `django-allauth` para la autenticación, hay algunas cosas que debes tener en cuenta con respecto al modelo `User`:

1. **Configuración de `django-allauth`:** Debes asegurarte de que `django-allauth` esté correctamente configurado en tu proyecto. Esto incluye agregar `allauth` y `allauth.account` a tu `INSTALLED_APPS` en `settings.py`, configurar tu `AUTHENTICATION_BACKENDS` para incluir `allauth.account.auth_backends.AuthenticationBackend`, y agregar las URLs de `allauth` a tu `urls.py`.
2. **Campos de usuario:** `django-allauth` utiliza el modelo `User` de Django por defecto, que incluye campos para el nombre de usuario, el correo electrónico, la contraseña, el nombre y los apellidos. Si necesitas campos adicionales para tus usuarios, puedes crear un modelo de usuario personalizado que herede de `AbstractUser` o `AbstractBaseUser`. Sin embargo, debes tener en cuenta que `django-allauth` tiene ciertas expectativas sobre los campos disponibles en tu modelo de usuario, especialmente en lo que respecta al correo electrónico.
3. **Correo electrónico como nombre de usuario:** Si prefieres que tus usuarios se autenticquen con su correo electrónico en lugar de un nombre de usuario, puedes configurar `django-allauth` para que lo haga. Para hacer esto, debes establecer `ACCOUNT_AUTHENTICATION_METHOD` a `'email'` y `ACCOUNT_USERNAME_REQUIRED` a `False` en tu `settings.py`.
4. **Integración con el modelo `Profile`:** Si estás utilizando un modelo `Profile` para almacenar información adicional sobre tus usuarios, puedes configurar una señal en Django para crear automáticamente un perfil vacío cada vez que se crea un nuevo usuario. Esto asegura que cada usuario tenga un perfil asociado desde el momento en que se registran.

Aquí hay un ejemplo de cómo podrías configurar esta señal:

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth.models import User
from .models import Profile
@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
@receiver(post_save, sender=User)
def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()
```

En este ejemplo, `create_user_profile` se llama cada vez que se crea un nuevo `User`, y crea un `Profile` asociado. `save_user_profile` se llama cada vez que se guarda un `User`, y guarda el `Profile` asociado.

▼ Haystack search engine

Django Haystack provides modular search for Django. It abstracts the complexities of the various search backends while providing a consistent API to your code. Here's a basic example of how you can implement it:

1. **Install Django Haystack and a search engine:** You can install Django Haystack with pip. You'll also need to install a search engine. For this example, we'll use Whoosh, which is pure Python and easy to install, but not as powerful as others like Elasticsearch.

```
pip install django-haystack whoosh
```

2.

Update your settings: Add `'haystack'` to your `INSTALLED_APPS` and set `HAYSTACK_CONNECTIONS`.

```
INSTALLED_APPS = [
    # Other apps...
    'haystack',
]

HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/path/to/whoosh/index',
    },
}
```

3. **Create a search index:** A search index is a way of telling Haystack what data should be made searchable. For example, if you want to make your `Recipe` model searchable, you could create a `RecipeIndex`.

```
from haystack import indexes
from .models import Recipe

class RecipeIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)

    def get_model(self):
        return Recipe
```

```
def index_queryset(self, using=None):
    return self.get_model().objects.all()
```

In this example, `text` is the main search field. `use_template=True` means that Haystack will use a data template to build the document that will be indexed.

4. **Create a data template:** This is a text file that describes the data you're indexing. It should be named `<app_label>/<model_name>_text.txt`. For example, if your `Recipe` model is in an app called `recipes`, you would create a file called `recipes/recipe_text.txt`.

```
{{ object.name }}
{{ object.description }}
```

5.

Update your views and templates: You can now use Haystack's `SearchView` and `SearchQuerySet` in your views and templates.

```
from haystack.generic_views import SearchView
from haystack.query import SearchQuerySet

class RecipeSearchView(SearchView):
    template_name = 'search/search.html'
    queryset = SearchQuerySet().models(Recipe)
```

```
{% for result in page.object_list %}
    <h2><a href="{{ result.object.get_absolute_url }}">{{ result.object.name }}</a></h2>
    <p>{{ result.object.description }}</p>
{% empty %}
    <p>No results found.</p>
{% endfor %}
```

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

urlpatterns = [
    path('search/', RecipeSearchView.as_view(), name='search'),
    # Other URL patterns...
]
```

1. **Rebuild the search index:** Whenever you add or change data that's indexed, you need to rebuild the search index.

```
python manage.py rebuild_index
```

This is a very basic implementation of Django Haystack. Depending on your needs, you might want to customize this. For example, you might want to add more fields to the search index, customize the search results, etc.

▼ Forms

▼ Minimal form tutorial

Write a minimal form

Let's update our poll detail template ("polls/detail.html") from the last tutorial, so that the template contains an HTML `<form>` element:

```
polls/templates/polls/detail.html
< form action="{% url 'polls:vote' question.id %}" method="post">
{%
csrf_token %}
<
```

```

fieldset >
<
legend >< h1 >{{ question.question_text }}</ h1 ></ legend >
{%
if error_message %}< p >< strong >{{ error_message }}</ strong ></ p >{% endif %}
{%
for choice in question.choice_set.all %}
<
input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}">
<
label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</ label >< br >
{%
endfor %}
</
fieldset >
<
input type="submit" value="Vote">
</
form >

```

A quick rundown:

- The above template displays a radio button for each question choice. The `value` of each radio button is the associated question choice's ID. The `name` of each radio button is `"choice"`. That means, when somebody selects one of the radio buttons and submits the form, it'll send the POST data `choice=#` where `#` is the ID of the selected choice. This is the basic concept of HTML forms.
- We set the form's `action` to `{% url 'polls:vote' question.id %}`, and we set `method="post"`. Using `method="post"` (as opposed to `method="get"`) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use `method="post"`. This tip isn't specific to Django; it's good web development practice in general.
- `forloop.counter` indicates how many times the `for` tag has gone through its loop
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a helpful system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the `{% csrf token %}` template tag.

Now, let's create a Django view that handles the submitted data and does something with it. Remember, in [Tutorial 3](#), we created a URLconf for the polls application that includes this line:

```

polls/urls.py
path("<int:question_id>/vote/", views.vote, name="vote"),

```

We also created a dummy implementation of the `vote()` function. Let's create a real version. Add the following to `polls/views.py`:

```

polls/views.py
from django.db.models import F
from django.http import HttpResponseRedirect, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question

# ... def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)

    try :
        selected_choice = question.choice_set.get(pk=request.POST["choice"])

    except ( KeyError , Choice.DoesNotExist):

    # Redisplay the question voting form. return render(
        request,

```



```

        "polls/detail.html",
        {
            "question": question,
            "error_message": "You didn't select a choice.",
        },
    ),
)
else :
    selected_choice.votes = F("votes") + 1
    selected_choice.save()

# Always return an HttpResponseRedirect after successfully dealing # with POST data. This prevents data from being posted
# twice if a # user hits the Back button. return HttpResponseRedirect(reverse("polls:results", args=(question.id,)))

```

This code includes a few things we haven't covered yet in this tutorial:

- `request.POST` is a dictionary-like object that lets you access submitted data by key name. In this case, `request.POST['choice']` returns the ID of the selected choice, as a string. `request.POST` values are always strings.
- Note that Django also provides `request.GET` for accessing GET data in the same way – but we're explicitly using `request.POST` in our code, to ensure that data is only altered via a POST call.
- `request.POST['choice']` will raise `KeyError` if `choice` wasn't provided in POST data. The above code checks for `KeyError` and redisplay the question form with an error message if `choice` isn't given.
- `F("votes") + 1` instructs the database to increase the vote count by 1.
- After incrementing the choice count, the code returns an `HttpResponseRedirect` rather than a normal `HttpResponse`. `HttpResponseRedirect` takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

As the Python comment above points out, you should always return an `HttpResponseRedirect` after successfully dealing with POST data. This tip isn't specific to Django; it's good web development practice in general.

- We are using the `reverse()` function in the `HttpResponseRedirect` constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in [Tutorial 3](#), this `reverse()` call will return a string like

```
"/polls/3/results/"
```

where the `3` is the value of `question.id`. This redirected URL will then call the `'results'` view to display the final page.

As mentioned in [Tutorial 3](#), `request` is an `HttpRequest` object. For more on `HttpRequest` objects, see the [request and response documentation](#).

After somebody votes in a question, the `vote()` view redirects to the results page for the question. Let's write that view:

```

polls/views.py
from django.shortcuts import get_object_or_404, render

def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)

    return render(request, "polls/results.html", {"question": question})

```

This is almost exactly the same as the `detail()` view from [Tutorial 3](#). The only difference is the template name. We'll fix this redundancy later.

Now, create a `polls/results.html` template:

```

polls/templates/polls/results.html
< h1 >{{ question.question_text }}</ h1 >
<

```

```

ul >
{%
for choice in question.choice_set.all %}
<
li >{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</ li >
{%
endfor %}
</
ul >

<
a href="{% url 'polls:detail' question.id %}">Vote again?</ a >

```

Now, go to `/polls/1/` in your browser and vote in the question. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

▼ Model forms

Working with forms¶

About this document

This document provides an introduction to the basics of web forms and how they are handled in Django. For a more detailed look at specific areas of the forms API, see [The Forms API](#), [Form fields](#), and [Form and field validation](#).

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

HTML forms¶

In HTML, a form is a collection of elements inside `<form>...</form>` that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form `<input>` elements to achieve these effects.

As well as its `<input>` elements, a form must specify two things:

- *where*: the URL to which the data corresponding to the user's input should be returned
- *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several `<input>` elements: one of `type="text"` for the username, one of `type="password"` for the password, and one of `type="submit"` for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the `<form>`'s `action` attribute - `/admin/` - and that it should be sent using the HTTP mechanism specified by the `method` attribute - `post`.

When the `<input type="submit" value="Log in">` element is triggered, the data is returned to `/admin/`.

GET and POST ¶

`GET` and `POST` are the only HTTP methods to use when dealing with forms.

Django's login form is returned using the `POST` method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

`GET`, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form `https://docs.djangoproject.com/search?q=forms&release=1`.

`GET` and `POST` are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use `POST`. `GET` should be used only for requests that do not affect the state of the system.

`GET` would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A web application that uses `GET` requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system. `POST`, coupled with other protections like Django's [CSRF protection](#) offers more control over access.

On the other hand, `GET` is suitable for things like a web search form, because the URLs that represent a `GET` request can easily be bookmarked, shared, or resubmitted.

Django's role in forms

Handling forms is a complex business. Consider Django's admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

Forms in Django

We've described HTML forms briefly, but an HTML `<form>` is just one part of the machinery required.

In the context of a web application, 'form' might refer to that HTML `<form>`, or to the Django `Form` that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

The Django `Form` class

At the heart of this system of components is Django's `Form` class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a `Form` class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form `<input>` elements. (A `ModelForm` maps a model class's fields to HTML form `<input>` elements via a `Form`; this is what the Django admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A `DateField` and a `FileField` handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML “widget” - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Instantiating, processing, and rendering forms¶

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that’s what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we’re dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or prepopulate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

The last of these cases is the most interesting, because it’s what makes it possible for users not just to read a website, but to send information back to it too.

Building a form¶

The work that needs to be done¶

Suppose you want to create a simple form on your website, in order to obtain the user’s name. You’d need something like this in your template:

```
< form action="/your-name/" method="post">
<
label for="your_name">Your name: </ label >
<
input id="your_name" type="text" name="your_name" value="{{ current_name }}">
<
input type="submit" value="OK">
</
form >
```

This tells the browser to return the form data to the URL `/your-name/`, using the `POST` method. It will display a text field, labeled “Your name:”, and a button marked “OK”. If the template context contains a `current_name` variable, that will be used to pre-fill the `your_name` field.

You’ll need a view that renders the template containing the HTML form, and that can supply the `current_name` field as appropriate.

When the form is submitted, the `POST` request which is sent to the server will contain the form data.

Now you’ll also need a view corresponding to that `/your-name/` URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be prepopulated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

Building a form in Django

The `Form` class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

```
forms.py
from django import forms

class NameForm (forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

This defines a `Form` class with a single field (`your_name`). We've applied a human-friendly label to the field, which will appear in the `<label>` when it's rendered (although in this case, the `label` we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by `max_length`. This does two things. It puts a `maxlength="100"` on the HTML `<input>` (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A `Form` instance has an `is_valid()` method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return `True`
- place the form's data in its `cleaned_data` attribute.

The whole form, when rendered for the first time, will look like:

```
< label for="your_name">Your name: </ label >
<
input id="your_name" type="text" name="your_name" maxlength="100" required>
```

Note that it **does not** include the `<form>` tags, or a submit button. We'll have to provide those ourselves in the template.

The view

Form data sent back to a Django website is processed by a view, generally the same view which published the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the view for the URL where we want it to be published:

```
views.py
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data if request.method == "POST":
    # create a form instance and populate it with data from the request: form = NameForm(request.POST)
    # check whether it's valid: if form.is_valid():
    # process the data in form.cleaned_data as required # ... # redirect to a new URL: return
```

```
HttpResponseRedirect("/thanks/")
```

```
# if a GET (or any other method) we'll create a blank form else :  
form = NameForm()
```

```
return render(request, "name.html", {"form": form})
```

If we arrive at this view with a `GET` request, it will create an empty form instance and place it in the template context to be rendered. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a `POST` request, the view will once again create a form instance and populate it with data from the request: `form = NameForm(request.POST)` This is called “binding data to the form” (it is now a *bound* form).

We call the form’s `is_valid()` method; if it’s not `True`, we go back to the template with the form. This time the form is no longer empty (*unbound*) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If `is_valid()` is `True`, we’ll now be able to find all the validated form data in its `cleaned_data` attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

The template¶

We don’t need to do much in our `name.html` template:

```
< form action="/your-name/" method="post">  
  {%  
    csrf_token %}  
  {{ form }}  
  <  
    input type="submit" value="Submit">  
  </  
form >
```

All the form’s fields and their attributes will be unpacked into HTML markup from that `{{ form }}` by Django’s template language.

Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use [protection against Cross Site Request Forgeries](#). When submitting a form via `POST` with CSRF protection enabled you must use the `csrf_token` template tag as in the preceding example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

HTML5 input types and browser validation

If your form includes a `URLField`, an `EmailField` or any integer field type, Django will use the `url`, `email` and `number` HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Django’s validation. If you would like to disable this behavior, set the `novalidate` attribute on the `form` tag, or specify a different widget on the field, like `TextInput`.

We now have a working web form, described by a Django `Form`, processed by a view, and rendered as an HTML `<form>`.

That’s all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

More about Django `Form` classes¶

All form classes are created as subclasses of either `django.forms.Form` or `django.forms.ModelForm`. You can think of `ModelForm` as a subclass

of `Form`, `Form` and `ModelForm` actually inherit common functionality from a (private) `BaseForm` class, but this implementation detail is rarely important.

Models and Forms

In fact if your form is going to be used to directly add or edit a Django model, a `ModelForm` can save you a great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their attributes, from a `Model` class.

Bound and unbound form instances¶

The distinction between Bound and unbound forms is important:

- An unbound form has no data associated with it. When rendered to the user, it will be empty or will contain default values.
- A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form's `is_bound` attribute will tell you whether a form has data bound to it or not.

More on fields¶

Consider a more useful form than our minimal example above, which we could use to implement “contact me” functionality on a personal website:

`forms.py`¶

```
from django import forms
```

```
class ContactForm (forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False )
```

Our earlier form used a single field, `your_name`, a `CharField`. In this case, our form has four fields: `subject`, `message`, `sender` and `cc_myself`. `CharField`, `EmailField` and `BooleanField` are just three of the available field types; a full list can be found in Form fields.

Widgets¶

Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as `<input type="text">`.

In most cases, the field will have a sensible default widget. For example, by default, a `CharField` will have a `TextInput` widget, that produces an `<input type="text">` in the HTML. If you needed `<textarea>` instead, you'd specify the appropriate widget when defining your form field, as we have done for the `message` field.

Field data¶

Whatever the data submitted with a form, once it has been successfully validated by calling `is_valid()` (and `is_valid()` has returned `True`), the validated form data will be in the `form.cleaned_data` dictionary. This data will have been nicely converted into Python types for you.

Note

You can still access the unvalidated data directly from `request.POST` at this point, but the validated data is better.

In the contact form example above, `cc_myself` will be a boolean value. Likewise, fields such as `IntegerField` and `FloatField` convert values to a Python `int` and `float` respectively. Here's how the form data could be processed in the view that handles this form:


```
views.py:1
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data["subject"]
    message = form.cleaned_data["message"]
    sender = form.cleaned_data["sender"]
    cc_myself = form.cleaned_data["cc_myself"]

    recipients = ["info@example.com"]

    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)

    return HttpResponseRedirect("/thanks/")
```

Tip

For more on sending email from Django, see [Sending email](#).

Some field types need some extra handling. For example, files that are uploaded using a form need to be handled differently (they can be retrieved from `request.FILES`, rather than `request.POST`). For details of how to handle file uploads with your form, see [Binding uploaded files to a form](#).

Working with form templates

All you need to do to get your form into a template is to place the form instance into the template context. So if your form is called `form` in the context, `{{ form }}` will render its `<label>` and `<input>` elements appropriately.

Additional form template furniture

Don't forget that a form's output does *not* include the surrounding `<form>` tags, or the form's `submit` control. You will have to provide these yourself.

Reusable form templates

The HTML output when rendering a form is itself generated via a template. You can control this by creating an appropriate template file and setting a custom `FORM_RENDERER` to use that `form_template_name` site-wide. You can also customize per-form by overriding the form's `template_name` attribute to render the form using the custom template, or by passing the template name directly to `Form.render()`.

The example below will result in `{{ form }}` being rendered as the output of the `form_snippet.html` template.

In your templates:

```
# In your template:
{{ form }}

# In form_snippet.html:
{%
for field in form %}
    <
    div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{%
endfor %}
```

Then you can configure the `FORM_RENDERER` setting:

```
settings.py:1
from django.forms.renderers import TemplatesSetting

class CustomFormRenderer(TemplatesSetting):
    form_template_name = "form_snippet.html"
```

```
FORM_RENDERER = "project.settings.CustomFormRenderer"
```

... or for a single form:

```
class MyForm (forms.Form):
    template_name = "form_snippet.html"
    ...
```

... or for a single render of a form instance, passing in the template name to the `Form.render()`. Here's an example of this being used in a view:

```
def index(request):
    form = MyForm()
    rendered_form = form.render("form_snippet.html")
    context = {"form": rendered_form}

    return render(request, "index.html", context)
```

See [Outputting forms as HTML](#) for more details.

Reusable field group templates¶

New in Django 5.0.

Each field is available as an attribute of the form, using `{{ form.name_of_field }}` in a template. A field has a `as_field_group()` method which renders the related elements of the field as a group, its label, widget, errors, and help text.

This allows generic templates to be written that arrange fields elements in the required layout. For example:

```
{{ form.non_field_errors }}
<


By default Django uses the "django/forms/field.html" template which is designed for use with the default "django/forms/div.html" form style.



The default template can be customized by setting field_template_name in your project-level FORM_RENDERER:



```
from django.forms.renderers import TemplatesSetting
```



```
class CustomFormRenderer (TemplatesSetting):
 field_template_name = "field_snippet.html"
```



... or on a single field:



```
class MyForm (forms.Form):
 subject = forms.CharField(template_name="my_custom_template.html")
 ...
```



... or on a per-request basis by calling BoundField.render() and supplying a template name:



```
def index(request):
 form = ContactForm()
 subject = form["subject"]
 context = {"subject": subject.render("my_custom_template.html")}

 return render(request, "index.html", context)
```



## Rendering fields manually¶



Django - Python



24


```

More fine grained control over field rendering is also possible. Likely this will be in a custom field template, to allow the template to be written once and reused for each field. However, it can also be directly accessed from the field attribute on the form. For example:

```
{{ form.non_field_errors }}
<
div class="fieldWrapper">
    {{ form.subject.errors }}
    <
    label for="{{ form.subject.id_for_label }}">Email subject:</ label >
    {{ form.subject }}
</
div >
<
div class="fieldWrapper">
    {{ form.message.errors }}
    <
    label for="{{ form.message.id_for_label }}">Your message:</ label >
    {{ form.message }}
</
div >
<
div class="fieldWrapper">
    {{ form.sender.errors }}
    <
    label for="{{ form.sender.id_for_label }}">Your email address:</ label >
    {{ form.sender }}
</
div >
<
div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <
    label for="{{ form.cc_myself.id_for_label }}">CC yourself?</ label >
    {{ form.cc_myself }}
</
div >
```

Complete `<label>` elements can also be generated using the `label_tag()`. For example:

```
< div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</
div >
```

Rendering form error messages¶

The price of this flexibility is a bit more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for each field and any errors for the form as a whole. Note `{{ form.non_field_errors }}` at the top of the form and the template lookup for errors on each field.

Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
< ul class="errorlist">
    <
    li >Sender is required.</ li >
</
ul >
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}
<
ol >
    {%
    for error in form.subject.errors %}
    <
    li >< strong >{{ error|escape }}</ strong ></ li >
    {%
    endfor %}
</
ol >
{%
endif %}
```

```
< ul class="errorlist nonfield">
  <
  li >Generic validation error</ li >
</
ul >
```

Looping over the form's fields¶

```
{% for field in form %}
  <
  div class="fieldWrapper">
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
    {%
    if field.help_text %}
      <
      p class="help" id="{{ field.auto_id }}_helptext">
        {{ field.help_text|safe }}
      </
    p >
    {%
  endif %}
  </
  div >
  {%
endfor %}
```

`{{ field.errors }}` Outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a string containing the error message. `{{ field.field }}` The `Field` instance from the form class that this `BoundField` wraps. You can use it to access `Field` attributes, e.g. `{{ char_field.field.max_length }}`. `{{ field.help_text }}` Any help text that has been associated with the field. `{{ field.html_name }}` The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set. `{{ field.id_for_label }}` The ID that will be used for this field (`id_email` in the example above). If you are constructing the label manually, you may want to use this in lieu of `label_tag`. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID. `{{ field.is_hidden }}` This attribute is `True` if the form field is a hidden field and `False` otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

`{{ field.label }}` The label of the field, e.g. `Email address`. `{{ field.label_tag }}` The field's label wrapped in the appropriate HTML `<label>` tag. This includes the form's `label_suffix`. For example, the default `label_suffix` is a colon:

Similar to `field.label_tag` but uses a `<legend>` tag in place of `<label>`, for widgets with multiple inputs wrapped in a `<fieldset>`.

```
{{ field.use_fieldset }}
```

This attribute is True if the form field's widget contains multiple inputs that should be semantically grouped in a `<fieldset>` with a `<legend>` to improve accessibility. An example use in a template:

```
{% if field.use_fieldset %}
<
fieldset >
{%
if field.label %}{{ field.legend_tag }}{% endif %}
{%
else %}
{%
if field.label %}{{ field.label_tag }}{% endif %}
{%
endif %}
{{ field }}
{%
if field.use_fieldset %}</ fieldset >{% endif %}
{{ field.value }} The value of the field. e.g someone@example.com.
```

See also

For a complete list of attributes and methods, see [RoundField](#).

Looping over hidden and visible fields¶

If you're manually laying out a form in a template, as opposed to relying on Django's default form layout, you might want to treat `<input type="hidden">` fields differently from non-hidden fields. For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Django provides two methods on a form that allow you to loop over the hidden and visible fields independently: `hidden_fields()` and `visible_fields()`. Here's a modification of an earlier example that uses these two methods:

```
{# Include the hidden fields #} {% for hidden in form.hidden_fields %}
{{ hidden }}
{%
endfor %}

{# Include the visible fields #} {% for field in form.visible_fields %}
<
div class="fieldWrapper">
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
</
div >
{%
endfor %}
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.

▼ Admin tutorial

Writing your first Django app, part 7¶

This tutorial begins where [Tutorial 6](#) left off. We're continuing the web-poll application and will focus on customizing Django's automatically-generated admin site that we first explored in [Tutorial 2](#).

Where to get help:

If you're having trouble going through this tutorial, please head over to the [Getting_Help](#) section of the FAQ.

Customize the admin form¶

By registering the `Question` model with `admin.site.register(Question)`, Django was able to construct a default form representation. Often, you'll want to customize how the admin form looks and works. You'll do this by telling Django the options you want when you register the object.

Let's see how this works by reordering the fields on the edit form. Replace the `admin.site.register(Question)` line with:

```
polls/admin.py¶
from django.contrib import admin

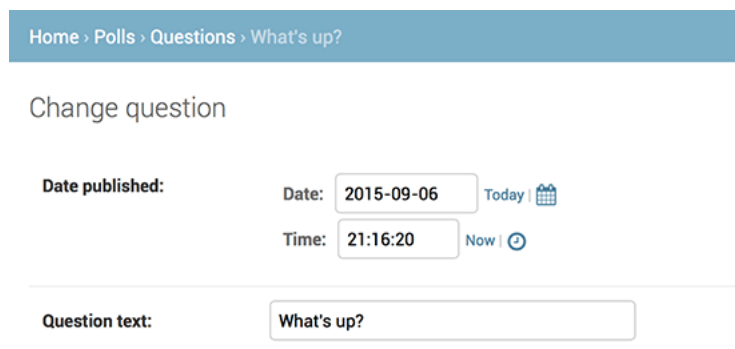
from .models import Question

class QuestionAdmin (admin.ModelAdmin):
    fields = ["pub_date", "question_text"]

admin.site.register(Question, QuestionAdmin)
```

You'll follow this pattern – create a model admin class, then pass it as the second argument to `admin.site.register()` – any time you need to change the admin options for a model.

This particular change above makes the “Publication date” come before the “Question” field:



This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

And speaking of forms with dozens of fields, you might want to split the form up into fieldsets:

```
polls/admin.py¶
from django.contrib import admin

from .models import Question

class QuestionAdmin (admin.ModelAdmin):
    fieldsets = [
        (None, {"fields": ["question_text"]}),
        ("Date information", {"fields": ["pub_date"]}),
    ]

admin.site.register(Question, QuestionAdmin)
```

The first element of each tuple in `fieldsets` is the title of the fieldset. Here's what our form looks like now:

Home > Polls > Questions > What's up?

Change question

Question text:

What's up?

Date information

Date published:

Date: 2015-09-06

Today 

Time: 21:16:20

Now 

Adding related objects¶

OK, we have our Question admin page, but a `Question` has multiple `Choices`, and the admin page doesn't display choices.

Yet.

There are two ways to solve this problem. The first is to register `Choice` with the admin just as we did with `Question`:

```
polls/admin.py¶  
  
from django.contrib import admin  
  
from .models import Choice, Question  
  
# ... admin.site.register(Choice)
```

Now "Choices" is an available option in the Django admin. The "Add choice" form looks like this:

Home > Polls > Choices > Add choice

Add choice

Question:

-----   

Choice text:

Votes:

0  

In that form, the "Question" field is a select box containing every question in the database. Django knows that a `ForeignKey` should be represented in the admin as a `<select>` box. In our case, only one question exists at this point.

Also note the "Add another question" link next to "Question." Every object with a `ForeignKey` relationship to another gets this for free. When you click "Add another question", you'll get a popup window with the "Add question" form. If you add a question in that window and click "Save", Django will save the question to the database and dynamically add it as the selected choice on the "Add choice" form you're looking at.

But, really, this is an inefficient way of adding `Choice` objects to the system. It'd be better if you could add a bunch of Choices directly when you create the `Question` object. Let's make that happen.

Remove the `register()` call for the `Choice` model. Then, edit the `Question` registration code to read:

```
polls/admin.py
from django.contrib import admin

from .models import Choice, Question

class ChoiceInline (admin.StackedInline):
    model = Choice
    extra = 3

class QuestionAdmin (admin.ModelAdmin):
    fieldsets = [
        (
            None, {"fields": ["question_text"]},
            ("Date information", {"fields": ["pub_date"], "classes": ["collapse"]}),
        )
    ]
    inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)
```

This tells Django: “`Choice` objects are edited on the `Question` admin page. By default, provide enough fields for 3 choices.”

Load the “Add question” page to see how that looks:

Add question

Question text:

Date information (Hide)

Date published:

Date: Today | Time: Now | 

CHOICES

Choice: #1 

Choice text:

Votes:

 Choice: #2 

Choice text:

Votes:

 Choice: #3 

Choice text:

Votes:

 [+ Add another Choice](#)[Save and add another](#)[Save and continue editing](#)[SAVE](#)

It works like this: There are three slots for related Choices – as specified by `extra` – and each time you come back to the “Change” page for an already-created object, you get another three extra slots.

At the end of the three current slots you will find an “Add another Choice” link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. This image shows an added slot:

CHOICES	
Choice: #1	
Choice text:	<input type="text"/>
Votes:	0
Choice: #2	
Choice text:	<input type="text"/>
Votes:	0
Choice: #3	
Choice text:	<input type="text"/>
Votes:	0
Choice: #4	
Choice text:	<input type="text"/>
Votes:	0
+ Add another Choice	

One small problem, though. It takes a lot of screen space to display all the fields for entering related `Choice` objects. For that reason, Django offers a tabular way of displaying inline related objects. To use it, change the `ChoiceInline` declaration to read:

```
polls/admin.py
```

```
class ChoiceInline (admin.TabularInline): ...
```

With that `TabularInline` (instead of `StackedInline`), the related objects are displayed in a more compact, table-based format:

CHOICES		
CHOICE TEXT	VOTES	DELETE?
<input type="text"/>	0	
<input type="text"/>	0	
<input type="text"/>	0	
+ Add another Choice		
<div> Save and add another Save and continue editing SAVE </div>		

Note that there is an extra “Delete?” column that allows removing rows added using the “Add another Choice” button and rows that have already been saved.

Customize the admin change list

Now that the Question admin page is looking good, let’s make some tweaks to the “change list” page – the one that displays all the questions in the system.

Here’s what it looks like at this point:

Select question to change

ADD QUESTION +

Action: Go 0 of 1 selected☐ QUESTION☐ What's up?

1 question

By default, Django displays the `str()` of each object. But sometimes it'd be more helpful if we could display individual fields. To do that, use the `list_display` admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

```
polls/admin.py
class QuestionAdmin (admin.ModelAdmin):
    # ... list_display = ["question_text", "pub_date"]
```

For good measure, let's also include the `was_published_recently()` method from [Tutorial 2](#):

```
polls/admin.py
class QuestionAdmin (admin.ModelAdmin):
    # ... list_display = ["question_text", "pub_date", "was_published_recently"]
```

Now the question change list page looks like this:

Select question to change

Action: Go 0 of 1 selected

<input type="checkbox"/> QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/> What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

You can click on the column headers to sort by those values – except in the case of the `was_published_recently` header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for `was_published_recently` is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output.

You can improve that by using the `display()` decorator on that method (in `polls/models.py`), as follows:

```
polls/models.py
from django.contrib import admin

class Question (models.Model):
    # ... @admin.display(
        boolean=
    True ,
    ordering="pub_date",
    description="Published recently?"
    )
    def was_published_recently(self):
```

```

now = timezone.now()
return now - datetime.timedelta(days=1) <= self.pub_date <= now

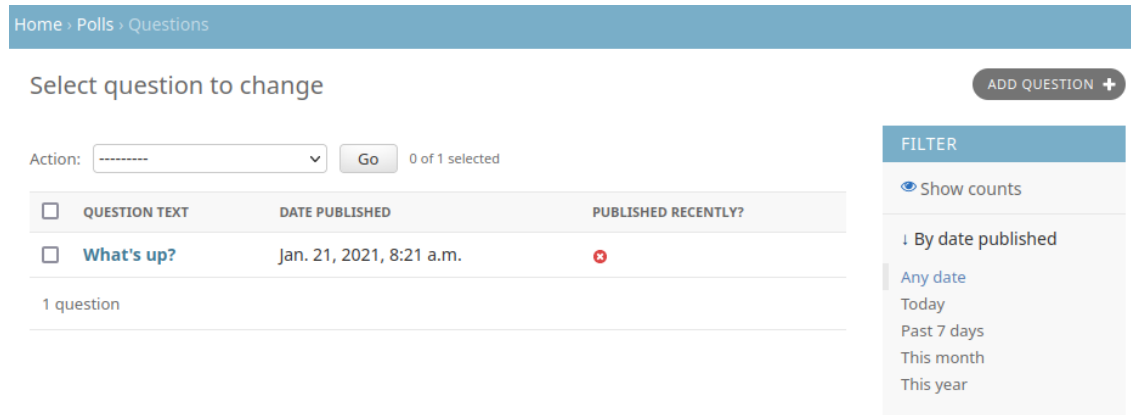
```

For more information on the properties configurable via the decorator, see [list_display](#).

Edit your `polls/admin.py` file again and add an improvement to the `Question` change list page: filters using the `list_filter`. Add the following line to `QuestionAdmin`:

```
list_filter = ["pub_date"]
```

That adds a “Filter” sidebar that lets people filter the change list by the `pub_date` field:



The type of filter displayed depends on the type of field you’re filtering on. Because `pub_date` is a `DateTimeField`, Django knows to give appropriate filter options: “Any date”, “Today”, “Past 7 days”, “This month”, “This year”.

This is shaping up well. Let’s add some search capability:

```
search_fields = ["question_text"]
```

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the `question_text` field. You can use as many fields as you’d like – although because it uses a `LIKE` query behind the scenes, limiting the number of search fields to a reasonable number will make it easier for your database to do the search.

Now’s also a good time to note that change lists give you free pagination. The default is to display 100 items per page. [Change list pagination](#), [search boxes](#), [filters](#), [date-hierarchies](#), and [column-header-ordering](#) all work together like you think they should.

Customize the admin look and feel¶

Clearly, having “Django administration” at the top of each admin page is ridiculous. It’s just placeholder text.

You can change it, though, using Django’s template system. The Django admin is powered by Django itself, and its interfaces use Django’s own template system.

Customizing your *project’s* templates¶

Create a `templates` directory in your project directory (the one that contains `manage.py`). Templates can live anywhere on your filesystem that Django can access. (Django runs as whatever user your server runs.) However, keeping your templates within the project is a good convention to follow.

Open your settings file (`mysite/settings.py`, remember) and add a `DIRS` option in the `TEMPLATES` setting:

```

mysite/settings.py¶

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS":

```

```
True,
"OPTIONS": {
    "context_processors": [
        "django.template.context_processors.debug",
        "django.template.context_processors.request",
        "django.contrib.auth.context_processors.auth",
        "django.contrib.messages.context_processors.messages",
    ],
},
],
]
```

`DIRS` is a list of filesystem directories to check when loading Django templates; it's a search path.

Organizing templates

Just like the static files, we *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application's template directory (e.g. `polls/templates`) rather than the project's (`templates`). We'll discuss in more detail in the [reusable apps tutorial](#) why we do this.

Now create a directory called `admin` inside `templates`, and copy the template `admin/base_site.html` from within the default Django admin template directory in the source code of Django itself (`django/contrib/admin/templates`) into that directory.

Where are the Django source files?

If you have difficulty finding where the Django source files are located on your system, run the following command:

```
❯ /❯
❯
❯ $ python -c "import django; print(django.__path__)"
```

Then, edit the file and replace `{{ site_header|default:('Django administration') }}` (including the curly braces) with your own site's name as you see fit. You should end up with a section of code like:

```
{% block branding %}
<
div id="site-name">< a href="{% url 'admin:index' %}">Polls Administration</ a ></ div >
{%
if user.is_anonymous %}
{%
include "admin/color_theme_toggle.html" %}
{%
endif %}
{%
endblock %}
```

We use this approach to teach you how to override templates. In an actual project, you would probably use the `django.contrib.admin.AdminSite.site_header` attribute to more easily make this particular customization.

This template file contains lots of text like `{% block branding %}` and `{{ title }}`. The `{%}` and `{{}}` tags are part of Django's template language. When Django renders `admin/base_site.html`, this template language will be evaluated to produce the final HTML page, just like we saw in [Tutorial 3](#).

Note that any of Django's default admin templates can be overridden. To override a template, do the same thing you did with `base_site.html` - copy it from the default directory into your custom directory, and make changes.

Customizing your application's templates¶

Astute readers will ask: But if `DIRS` was empty by default, how was Django finding the default admin templates? The answer is that, since `APP_DIRS` is set to `True`, Django automatically looks for a `templates/` subdirectory within each application package, for use as a fallback (don't forget that `django.contrib.admin` is an application).

Our poll application is not very complex and doesn't need custom admin templates. But if it grew more sophisticated and required modification of Django's standard admin

templates for some of its functionality, it would be more sensible to modify the *application's* templates, rather than those in the *project*. That way, you could include the polls application in any new project and be assured that it would find the custom templates it needed.

See the [template loading documentation](#) for more information about how Django finds its templates.

Customize the admin index page¶

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all the apps in `INSTALLED_APPS` that have been registered with the admin application, in alphabetical order. You may want to make significant changes to the layout. After all, the index is probably the most important page of the admin, and it should be easy to use.

The template to customize is `admin/index.html`. (Do the same as with `admin/base_site.html` in the previous section – copy it from the default directory to your custom template directory). Edit the file, and you'll see it uses a template variable called `app_list`. That variable contains every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best.

When you're comfortable with the admin, read [part 8 of this tutorial](#) to learn how to use third-party packages.

▼ Tests tutorial

Writing your first Django app, part 5¶

This tutorial begins where [Tutorial 4](#) left off. We've built a web-poll application, and we'll now create some automated tests for it.

Where to get help:

If you're having trouble going through this tutorial, please head over to the [Getting Help](#) section of the FAQ.

Introducing automated testing¶

What are automated tests?¶

Tests are routines that check the operation of your code.

Testing operates at different levels. Some tests might apply to a tiny detail (*does a particular model method return values as expected?*) while others examine the overall operation of the software (*does a sequence of user inputs on the site produce the desired result?*). That's no different from the kind of testing you did earlier in [Tutorial 2](#), using the `shell` to examine the behavior of a method, or running the application and entering data to check how it behaves.

What's different in *automated* tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your app, you can check that your code still works as you originally intended, without having to perform time consuming manual testing.

Why you need to create tests¶

So why create tests, and why now?

You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you

don't need to know how to create automated tests. But, if that's not the case, now is an excellent time to learn.

Tests will save you time¶

Up to a certain point, 'checking that it seems to work' will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them¶

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn't even realized it had gone wrong*.

Tests make your code more attractive¶

You might have created a brilliant piece of software, but you will find that many other developers will refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

Tests help teams work together¶

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that colleagues don't inadvertently break your code (and that you don't break theirs without knowing). If you want to make a living as a Django programmer, you must be good at writing tests!

Basic testing strategies¶

There are many ways to approach writing tests.

Some programmers follow a discipline called "test-driven development"; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development formalizes the problem in a Python test case.

More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it's never too late to get started.

Sometimes it's difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it's fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

So let's do that right away.

Writing our first test¶

We identify a bug¶

Fortunately, there's a little bug in the `polls` application for us to fix right away: the `Question.was_published_recently()` method returns `True` if the `Question` was published within the last day (which is correct) but also if the `Question`'s `pub_date` field is in the future (which certainly isn't).

Confirm the bug by using the `shell` to check the method on a question whose date lies in the future:

```
❯ /❯
```

```
❯
```

```
$ python manage.py shell
>>> import datetime >>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future >>> future_question = Question(pub_date=timezone.now()
+ datetime.timedelta(days=30))
>>> # was it published recently? >>> future_question.was_published_recently()
True
```

Since things in the future are not 'recent', this is clearly wrong.

Create a test to expose the bug¶

What we've just done in the `shell` to test for the problem is exactly what we can do in an automated test, so let's turn that into an automated test.

A conventional place for an application's tests is in the application's `tests.py` file; the testing system will automatically find tests in any file whose name begins with `test`.

Put the following in the `tests.py` file in the `polls` application:

```
polls/tests.py¶
import datetime from django.test import TestCase
from django.utils import timezone
from .models import Question

class QuestionModelTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future. """
        time =
        timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(),
        False )
```

Here we have created a `django.test.TestCase` subclass with a method that creates a `Question` instance with a `pub_date` in the future. We then check the output of `was_published_recently()` - which *ought* to be `False`.

Running tests¶

In the terminal, we can run our test:

0/0

0

```
$ python manage.py test polls
```

and you'll see something like:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
=====
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionModelTests)
-----
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16,
in test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False

-----
Ran 1 test
in 0.001s

FAILED (failures=1)
Destroying test database
for alias 'default'...
```

Different error?

If instead you're getting a `NameError` here, you may have missed a step in [Part 2](#) where we added imports of `datetime` and `timezone` to `polls/models.py`. Copy the imports from that section, and try running your tests again.

What happened is this:

- `manage.py test polls` looked for tests in the `polls` application
- it found a subclass of the `django.test.TestCase` class
- it created a special database for the purpose of testing
- it looked for test methods - ones whose names begin with `test`
- in `test_was_published_recently_with_future_question` it created a `Question` instance whose `pub_date` field is 30 days in the future
- ... and using the `assertIs()` method, it discovered that its `was_published_recently()` returns `True`, though we wanted it to return `False`

The test informs us which test failed and even the line on which the failure occurred.

Fixing the bug¶

We already know what the problem is: `Question.was_published_recently()` should return `False` if its `pub_date` is in the future. Amend the method in `models.py`, so that it will only return `True` if the date is also in the past:

```
polls/models.py¶
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

and run the test again:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

After identifying a bug, we wrote a test that exposes it and corrected the bug in the code so our test passes.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because running the test will warn us

immediately. We can consider this little portion of the application pinned down safely forever.

More comprehensive tests¶

While we're here, we can further pin down the `was_published_recently()` method; in fact, it would be positively embarrassing if in fixing one bug we had introduced another.

Add two more test methods to the same class, to test the behavior of the method more comprehensively:

```
polls/tests.py¶

def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is older than 1 day. """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(),
False )

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day. """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(),
True )
```

And now we have three tests that confirm that `Question.was_published_recently()` returns sensible values for past, recent, and future questions.

Again, `polls` is a minimal application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

Test a view¶

The polls application is fairly indiscriminating: it will publish any question, including ones whose `pub_date` field lies in the future. We should improve this. Setting a `pub_date` in the future should mean that the Question is published at that moment, but invisible until then.

A test for a view¶

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was an example of test-driven development, but it doesn't really matter in which order we do the work.

In our first test, we focused closely on the internal behavior of the code. For this test, we want to check its behavior as it would be experienced by a user through a web browser.

Before we try to fix anything, let's have a look at the tools at our disposal.

The Django test client¶

Django provides a test `Client` to simulate a user interacting with the code at the view level. We can use it in `tests.py` or even in the `shell`.

We will start again with the `shell`, where we need to do a couple of things that won't be necessary in `tests.py`. The first is to set up the test environment in the `shell`:

```
❏ /❏

❏

$ python manage.py shell

>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

`setup_test_environment()` installs a template renderer which will allow us to examine some additional attributes on responses such as `response.context` that otherwise wouldn't be available. Note that this method *does not* set up a test database, so the following will be run against the existing database and the output may differ slightly depending on what questions you already created. You might get unexpected results if your `TIME_ZONE` in `settings.py` isn't correct. If you don't remember setting it earlier, check it before continuing.

Next we need to import the test client class (later in `tests.py` we will use the `django.test.TestCase` class, which comes with its own client, so this won't be required):

```
>>> from django.test import Client
>>> # create an instance of the client for our use >>> client = Client()
```

With that ready, we can ask the client to do some work for us:

```
>>> # get a response from '/' >>> response = client.get("/")
Not Found: /

>>> # we should expect a 404 from that address; if you instead see an >>> # "Invalid HTTP_HOST header" error and a 400
response, you probably >>> # omitted the setup_test_environment() call described earlier. >>> response.status_code
404

>>> # on the other hand we should expect to find something at '/polls/' >>> # we'll use 'reverse()' rather than a hardcoded
URL >>> from django.urls import reverse

>>> response = client.get(reverse("polls:index"))

>>> response.status_code
200

>>> response.content
b'\n    <ul>\n        \n        <li><a href="/polls/1/">What&#x27;s up?</a></li>\n    \n    </ul>\n\n'

>>> response.context["latest_question_list"]
<QuerySet [<Question: What's up?>]>
```

Improving our view¶

The list of polls shows polls that aren't published yet (i.e. those that have a `pub_date` in the future). Let's fix that.

In [Tutorial 4](#) we introduced a class-based view, based on `ListView`:

```
polls/views.py¶

class IndexView (generic.ListView):
    template_name = "polls/index.html"
    context_object_name = "latest_question_list"

    def get_queryset(self):
        """Return the last five published questions.""" return Question.objects.order_by("-pub_date")[:5]
```

We need to amend the `get_queryset()` method and change it so that it also checks the date by comparing it with `timezone.now()`. First we need to add an import:

```
polls/views.py¶

from django.utils import timezone
```

and then we must amend the `get_queryset` method like so:

```
polls/views.py¶

def get_queryset(self):
    """Return the last five published questions (not including those set to be published in the future).""" return
    Question.objects.filter(pub_date__lte=timezone.now()).order_by("-pub_date")[:5]
]
```

`Question.objects.filter(pub_date__lte=timezone.now())` returns a queryset containing `Question`s whose `pub_date` is less than or equal to - that is, earlier than or equal to - `timezone.now`.

Testing our new view¶

Now you can satisfy yourself that this behaves as expected by firing up `runserver`, loading the site in your browser, creating `Questions` with dates in the past and future, and

checking that only those that have been published are listed. You don't want to have to do that *every single time you make any change that might affect this* - so let's also create a test, based on our `shell` session above.

Add the following to `polls/tests.py`:

```
polls/tests.py:1
from django.urls import reverse

and we'll create a shortcut function to create questions as well as a new test class:

polls/tests.py:1
def create_question(question_text, days):
    """
    Create a question with the given `question_text` and published the
    given number of `days` offset to now (negative
    for questions published in the past, positive for questions that have yet to be published).
    """
    time = timezone.now()
    + datetime.timedelta(days=days)

    return Question.objects.create(question_text=question_text, pub_date=time)

class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is displayed.
        """
        response = self.client.get(reverse("polls:index"))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerySetEqual(response.context["latest_question_list"], [])

    def test_past_question(self):
        """
        Questions with a pub_date in the past are displayed on the
        index page.
        """
        question = create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question],
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse("polls:index"))
        self.assertContains(response, "No polls are available.")
        self.assertQuerySetEqual(response.context["latest_question_list"], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are displayed.
        """
        question = create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question],
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        question1 = create_question(question_text="Past question 1.", days=-30)
        question2 = create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question2, question1],
        )
```

Let's look at some of these more closely.

First is a question shortcut function, `create_question`, to take some repetition out of the process of creating questions.

`test_no_questions` doesn't create any questions, but checks the message: "No polls are available." and verifies the `latest_question_list` is empty. Note that the `django.test.TestCase` class provides some additional assertion methods. In these examples, we use `assertContains()` and `assertQuerySetEqual()`.

In `test_past_question`, we create a question and verify that it appears in the list.

In `test_future_question`, we create a question with a `pub_date` in the future. The database is reset for each test method, so the first question is no longer there, and so again the index shouldn't have any questions in it.

And so on. In effect, we are using the tests to tell a story of admin input and user experience on the site, and checking that at every state and for every new change in the state of the system, the expected results are published.

Testing the `DetailView`

What we have works well; however, even though future questions don't appear in the `index`, users can still reach them if they know or guess the right URL. So we need to add a similar constraint to `DetailView`:

```
polls/views.py
class DetailView (generic.DetailView):
    ...

    def get_queryset(self):
        """ Excludes any questions that aren't published yet. """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

We should then add some tests, to check that a `Question` whose `pub_date` is in the past can be displayed, and that one with a `pub_date` in the future is not:

```
polls/tests.py
class QuestionDetailViewTests (TestCase):
    def test_future_question(self):
        """ The detail view of a question with a pub_date in the future returns a 404 not found. """
        future_question = create_question(question_text="Future question.", days=5)
        url = reverse("polls:detail", args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
        """ The detail view of a question with a pub_date in the past displays the question's text. """
        past_question = create_question(question_text="Past question.", days=-5)
        url = reverse("polls:detail", args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

Ideas for more tests

We ought to add a similar `get_queryset` method to `ResultsView` and create a new test class for that view. It'll be very similar to what we have just created; in fact there will be a lot of repetition.

We could also improve our application in other ways, adding tests along the way. For example, it's silly that `Questions` can be published on the site that have no `Choices`. So, our views could check for this, and exclude such `Questions`. Our tests would create a `Question` without `Choices` and then test that it's not published, as well as create a similar `Question` with `Choices`, and test that it is published.

Perhaps logged-in admin users should be allowed to see unpublished `Questions`, but not ordinary visitors. Again: whatever needs to be added to the software to accomplish this should be accompanied by a test, whether you write the test first and then make the code pass the test, or work out the logic in your code first and then write a test to prove it.

At a certain point you are bound to look at your tests and wonder whether your code is suffering from test bloat, which brings us to:

When testing, more is better¶

It might seem that our tests are growing out of control. At this rate there will soon be more code in our tests than in our application, and the repetition is unaesthetic, compared to the elegant conciseness of the rest of our code.

It doesn't matter. Let them grow. For the most part, you can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program.

Sometimes tests will need to be updated. Suppose that we amend our views so that only `Questions` with `Choices` are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves.

At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that's not a problem; in testing redundancy is a *good* thing.

As long as your tests are sensibly arranged, they won't become unmanageable. Good rules-of-thumb include having:

- a separate `TestClass` for each model or view
- a separate test method for each set of conditions you want to test
- test method names that describe their function

Further testing¶

This tutorial only introduces some of the basics of testing. There's a great deal more you can do, and a number of very useful tools at your disposal to achieve some very clever things.

For example, while our tests here have covered some of the internal logic of a model and the way our views publish information, you can use an "in-browser" framework such as [Selenium](#) to test the way your HTML actually renders in a browser. These tools allow you to check not just the behavior of your Django code, but also, for example, of your JavaScript. It's quite something to see the tests launch a browser, and start interacting with your site, as if a human being were driving it! Django includes `LiveServerTestCase` to facilitate integration with tools like Selenium.

If you have a complex application, you may want to run tests automatically with every commit for the purposes of [continuous integration](#), so that quality control is itself - at least partially - automated.

A good way to spot untested parts of your application is to check code coverage. This also helps identify fragile or even dead code. If you can't test a piece of code, it usually means that code should be refactored or removed. Coverage will help to identify dead code. See [Integration with coverage.py](#) for details.

[Testing in Django](#) has comprehensive information about testing.

▼ Shell tutorial

```
python manage.py shell
```

```
from polls.models import Choice, Question # Import the model classes we just wrote. # No
questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
```



```

# instead of datetime.datetime.now() and it will do the right thing.
>>>fromdjango.utilsimport timezone
>>>q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>>q.save()

# Now it has an ID.
>>>q.id
1

# Access model field values via Python attributes.
>>>q.question_text
"What's new?"
>>>q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=datetime.timezone.utc)

# Change values by changing the attributes, then calling save().
>>>q.question_text = "What's up?"
>>>q.save()

# objects.all() displays all the questions in the database.
>>>Question.objects.all()
<QuerySet [Question: Question object (1)]>

```

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

```

Creating objects¶

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call `save()` to save it to the database.

Assuming models live in a file `mysite/blog/models.py`, here's an example:

```

>>> from blog.models import Blog

>>> b = Blog(name="Beatles Blog", tagline="All the latest Beatles news.")

>>> b.save()

```

This performs an `INSERT` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

The `save()` method has no return value.

See also

`save()` takes a number of advanced options not described here. See the documentation for `save()` for complete details.

To create and save an object in a single step, use the `create()` method.

Saving changes to objects¶

To save changes to an object that's already in the database, use `save()`.

Given a `Blog` instance `b5` that has already been saved to the database, this example changes its name and updates its record in the database:

```

>>> b5.name = "New name"

>>> b5.save()

```

This performs an `UPDATE` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

Saving `ForeignKey` and `ManyToManyField` fields¶

Updating a `ForeignKey` field works exactly the same way as saving a normal field – assign an object of the right type to the field in question. This example updates the `blog` attribute of an `Entry` instance `entry`, assuming appropriate instances of `Entry` and `Blog` are already saved to the database (so we can retrieve them below):

```

>>> from blog.models import Blog, Entry

>>> entry = Entry.objects.get(pk=1)

>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")

>>> entry.blog = cheese_blog

>>> entry.save()

```

Updating a `ManyToManyField` works a little differently – use the `add()` method on the field to add a record to the relation. This example adds the `Author` instance `joe` to the `entry` object:

```

>>> from blog.models import Author

>>> joe = Author.objects.create(name="Joe")

```

```
>>> entry.authors.add(joe)
```

To add multiple records to a `ManyToManyField` in one go, include multiple arguments in the call to `add()`, like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

Retrieving objects¶

To retrieve objects from your database, construct a `QuerySet` via a `Manager` on your model class.

A `QuerySet` represents a collection of objects from your database. It can have zero, one or many *filters*. Filters narrow down the query results based on the given parameters. In SQL terms, a `QuerySet` equates to a `SELECT` statement, and a filter is a limiting clause such as `WHERE` or `LIMIT`.

You get a `QuerySet` by using your model's `Manager`. Each model has at least one `Manager`, and it's called `objects` by default. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name="Foo", tagline="Bar")
>>> b.objects
Traceback:
...
AttributeError: "Manager isn't accessible via Blog instances."
```

Note

`Managers` are accessible only via model classes, rather than from model instances, to enforce a separation between “table-level” operations and “record-level” operations.

The `Manager` is the main source of `QuerySets` for a model. For example, `Blog.objects.all()` returns a `QuerySet` that contains all `Blog` objects in the database.

Retrieving all objects¶

The simplest way to retrieve objects from a table is to get all of them. To do this, use the `all()` method on a `Manager`:

```
>>> all_entries = Entry.objects.all()
```

The `all()` method returns a `QuerySet` of all the objects in the database.

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. The two most common ways to refine a `QuerySet` are:

`filter(**kwargs)` Returns a new `QuerySet` containing objects that match the given lookup parameters. `exclude(**kwargs)` Returns a new `QuerySet` containing objects that do *not* match the given lookup parameters.

The lookup parameters (`**kwargs` in the above function definitions) should be in the format described in [Field lookups](#) below.

For example, to get a `QuerySet` of blog entries from the year 2006, use `filter()` like so:

```
Entry.objects.filter(pub_date__year=2006)
```

With the default manager class, it is the same as:

```
Entry.objects.all().filter(pub_date__year=2006)
```

Chaining filters¶

The result of refining a `QuerySet` is itself a `QuerySet`, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(headline__startswith="What").exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(pub_date__gte=datetime.date(2005, 1, 30))
```

This takes the initial `QuerySet` of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a `QuerySet` containing all entries with a headline that starts with “What”, that were published between January 30, 2005, and the current day.

Filtered `QuerySet`s are unique¶

Each time you refine a `QuerySet`, you get a brand-new `QuerySet` that is in no way bound to the previous `QuerySet`. Each refinement creates a separate and distinct `QuerySet` that can be stored, used and reused.

Example:

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

These three `QuerySets` are separate. The first is a base `QuerySet` containing all entries that contain a headline starting with “What”. The second is a subset of the first, with an additional criteria that excludes records whose `pub_date` is today or in the future. The third is a subset of the first, with an additional criteria that selects only the records whose `pub_date` is today or in the future. The initial `QuerySet` (`q1`) is unaffected by the refinement process.

▼ Django Rest Framework

▼ Install

Installation

Install using `pip`, including any optional packages you want...

```
pip install django-rest-framework
pip install markdown          # Markdown support for the browsable API.
pip install django-filter     # Filtering support
```

...or clone the project from github.

```
git clone https://github.com/encode/django-rest-framework
```

Add `'rest_framework'` to your `INSTALLED_APPS` setting.

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

If you're intending to use the browsable API you'll probably also want to add REST framework's login and logout views. Add the following to your root `urls.py` file.

```
urlpatterns = [
    ...
```

```
path('api-auth/', include('rest_framework.urls'))
]
```

Note that the URL path can be whatever you want.

Example

Let's take a look at a quick example of using REST framework to build a simple model-backed API.

We'll create a read-write API for accessing information on the users of our project.

Any global settings for a REST framework API are kept in a single configuration dictionary named `REST_FRAMEWORK`. Start off by adding the following to your `settings.py` module:

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Don't forget to make sure you've also added `rest_framework` to your `INSTALLED_APPS`.

We're ready to create our API now. Here's our project's root `urls.py` module:

```
from django.urls import path, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'is_staff']

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

You can now open the API in your browser at <http://127.0.0.1:8000/>, and view your new 'users' API. If you use the login control in the top right corner you'll also be able to add, create and delete users from the system.

▼ Example

Project setup

Create a new Django project named `tutorial`, then start a new app called `quickstart`.

```
# Create the project directory
mkdir tutorial
cd tutorial

# Create a virtual environment to isolate our package dependencies locally
python3 -m venv env
source env/bin/activate # On Windows use `env\Scripts\activate`

# Install Django and Django REST framework into the virtual environment
pip install django
pip install djangorestframework

# Set up a new project with a single application
django-admin startproject tutorial . # Note the trailing '.' character
cd tutorial
django-admin startapp quickstart
cd ..
```

The project layout should look like:

```
$ pwd
<some path>/tutorial
$ find .
.
./tutorial
./tutorial/asgi.py
./tutorial/__init__.py
./tutorial/quickstart
./tutorial/quickstart/migrations
./tutorial/quickstart/migrations/__init__.py
./tutorial/quickstart/models.py
./tutorial/quickstart/__init__.py
./tutorial/quickstart/apps.py
./tutorial/quickstart/admin.py
./tutorial/quickstart/tests.py
./tutorial/quickstart/views.py
./tutorial/settings.py
./tutorial/urls.py
./tutorial/wsgi.py
./env
./env/...
./manage.py
```

It may look unusual that the application has been created within the project directory. Using the project's namespace avoids name clashes with external modules (a topic that goes outside the scope of the quickstart).

Now sync your database for the first time:

```
python manage.py migrate
```

We'll also create an initial user named `admin` with a password. We'll authenticate as that user later in our example.

```
python manage.py createsuperuser --username admin --email admin@example.com
```

Once you've set up a database and the initial user is created and ready to go, open up the app's directory and we'll get coding...

Serializers

First up we're going to define some serializers. Let's create a new module named `tutorial/quickstart/serializers.py` that we'll use for our data representations.

```
from django.contrib.auth.models import Group, User
from rest_framework import serializers

class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'groups']

class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ['url', 'name']
```

Notice that we're using hyperlinked relations in this case with `HyperlinkedModelSerializer`. You can also use primary key and various other relationships, but hyperlinking is good RESTful design.

Views

Right, we'd better write some views then. Open `tutorial/quickstart/views.py` and get typing.

```
from django.contrib.auth.models import Group, User
from rest_framework import permissions, viewsets

from tutorial.quickstart.serializers import GroupSerializer, UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer
    permission_classes = [permissions.IsAuthenticated]

class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
    permission_classes = [permissions.IsAuthenticated]
```

Rather than write multiple views we're grouping together all the common behavior into classes called `ViewSet`s.

We can easily break these down into individual views if we need to, but using viewsets keeps the view logic nicely organized as well as being very concise.

URLs

Okay, now let's wire up the API URLs. On to `tutorial/urls.py`...

```
from django.urls import include, path
from rest_framework import routers

from tutorial.quickstart import views

router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]

urlpatterns += router.urls
```

Because we're using viewsets instead of views, we can automatically generate the URL conf for our API, by simply registering the viewsets with a router class.

Again, if we need more control over the API URLs we can simply drop down to using regular class-based views, and writing the URL conf explicitly.

Finally, we're including default login and logout views for use with the browsable API. That's optional, but useful if your API requires authentication and you want to use the browsable API.

Pagination

Pagination allows you to control how many objects per page are returned. To enable it add the following lines to `tutorial/settings.py`

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

Settings

Add `'rest_framework'` to `INSTALLED_APPS`. The settings module will be in `tutorial/settings.py`

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

Okay, we're done.

Testing our API

We're now ready to test the API we've built. Let's fire up the server from the command line.

```
python manage.py runserver
```

We can now access our API, both from the command-line, using tools like `curl`...

```
bash: curl -u admin -H 'Accept: application/json; indent=4' http://127.0.0.1:8000/users/
Enter host password for user 'admin':
{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "url": "http://127.0.0.1:8000/users/1/",
      "username": "admin",
      "email": "admin@example.com",
      "groups": []
    }
  ]
}
```

Or using the `httplib`, command line tool...

```
bash: http -a admin http://127.0.0.1:8000/users/
http: password for admin@127.0.0.1:8000::
$HTTP/1.1 200 OK
...
{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "email": "admin@example.com",
      "groups": [],
      "url": "http://127.0.0.1:8000/users/1/",
      "username": "admin"
    }
  ]
}
```

Or directly through the browser, by going to the URL `http://127.0.0.1:8000/users/`...

▼ Api requests

Request parsing

REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.

.data

`request.data` returns the parsed content of the request body. This is similar to the standard `request.POST` and `request.FILES` attributes except that:

- It includes all parsed content, including *file and non-file* inputs.
- It supports parsing the content of HTTP methods other than `POST`, meaning that you can access the content of `PUT` and `PATCH` requests.
- It supports REST framework's flexible request parsing, rather than just supporting form data. For example you can handle incoming `JSON data` similarly to how you handle incoming `form data`.

For more details see the [parsers documentation](#).

.query_params

`request.query_params` is a more correctly named synonym for `request.GET`.

For clarity inside your code, we recommend using `request.query_params` instead of the Django's standard `request.GET`. Doing so will help keep your codebase more correct and obvious - any HTTP method type may include query parameters, not just `GET` requests.

.parsers

The `APIView` class or `@api_view` decorator will ensure that this property is automatically set to a list of `Parser` instances, based on the `parser_classes` set on the view or based on the `DEFAULT_PARSER_CLASSES` setting.

You won't typically need to access this property.

Note: If a client sends malformed content, then accessing `request.data` may raise a `ParseError`. By default REST framework's `APIView` class or `@api_view` decorator will catch the error and return a `400 Bad Request` response.

If a client sends a request with a content-type that cannot be parsed then a `UnsupportedMediaType` exception will be raised, which by default will be caught and return a `415 Unsupported Media Type` response.

Content negotiation

The request exposes some properties that allow you to determine the result of the content negotiation stage. This allows you to implement behavior such as selecting a different serialization schemes for different media types.

.accepted_renderer

The renderer instance that was selected by the content negotiation stage.

.accepted_media_type

A string representing the media type that was accepted by the content negotiation stage.

Authentication

REST framework provides flexible, per-request authentication, that gives you the ability to:

- Use different authentication policies for different parts of your API.
- Support the use of multiple authentication policies.
- Provide both user and token information associated with the incoming request.

.user

`request.user` typically returns an instance of `django.contrib.auth.models.User`, although the behavior depends on the authentication policy being used.

If the request is unauthenticated the default value of `request.user` is an instance of `django.contrib.auth.models.AnonymousUser`.

For more details see the [authentication documentation](#).

.auth

`request.auth` returns any additional authentication context. The exact behavior of `request.auth` depends on the authentication policy being used, but it may typically be an instance of the token that the request was authenticated against.

If the request is unauthenticated, or if no additional context is present, the default value of `request.auth` is `None`.

For more details see the [authentication documentation](#).

.authenticators

The `APIView` class or `@api_view` decorator will ensure that this property is automatically set to a list of `Authentication` instances, based on the `authentication_classes` set on the view or based on the `DEFAULT_AUTHENTICATORS` setting.

You won't typically need to access this property.

Note: You may see a `WrappedAttributeError` raised when calling the `.user` or `.auth` properties. These errors originate from an authenticator as a standard `AttributeError`, however it's necessary that they be re-raised as a different exception type in order to prevent them from being suppressed by the outer property access. Python will not recognize that the `AttributeError` originates from the authenticator and will instead assume that the request object does not have a `.user` or `.auth` property. The authenticator will need to be fixed.

Browser enhancements

REST framework supports a few browser enhancements such as browser-based `PUT`, `PATCH` and `DELETE` forms.

.method

`request.method` returns the **uppercased** string representation of the request's HTTP method.

Browser-based `PUT`, `PATCH` and `DELETE` forms are transparently supported.

For more information see the [browser enhancements documentation](#).

.content_type

`request.content_type`, returns a string object representing the media type of the HTTP request's body, or an empty string if no media type was provided.

You won't typically need to directly access the request's content type, as you'll normally rely on REST framework's default request parsing behavior.

If you do need to access the content type of the request you should use the `.content_type` property in preference to using `request.META.get('HTTP_CONTENT_TYPE')`, as it provides transparent support for browser-based non-form content.

For more information see the [browser enhancements documentation](#).

.stream

`request.stream` returns a stream representing the content of the request body.

You won't typically need to directly access the request's content, as you'll normally rely on REST framework's default request parsing behavior.

Standard HttpRequest attributes

As REST framework's `Request` extends Django's `HttpRequest`, all the other standard attributes and methods are also available. For example the `request.META` and `request.session` dictionaries are available as normal.

Note that due to implementation reasons the `Request` class does not inherit from `HttpRequest` class, but instead extends the class using composition.

▼ Python

▼ Truquitos

▼ Comprobar si algo es iterable

(Útil para casos de array reduce etc)

`isinstance(variable, Iterable)` habiendo hecho `import` previo de `Iterable`

```
from collections import Iterable
lista = [1, 2, 3, 4]
cadena = "Python"
numero = 10
print(isinstance(lista, Iterable)) #True
print(isinstance(cadena, Iterable)) #True
print(isinstance(numero, Iterable)) #False
```

▼ Maneras eficaces de recorrer iterables

▼ List comprehension

▼ Desempaquetado

▼ *args y **kwargs

▼ _ variable

- `for _ in range(30)`: This is a for loop that will iterate 30 times. The underscore `_` is a common convention in Python for a variable that we don't plan on using. In this case, we don't care about the actual numbers 0 through 29 that `range(30)` generates, we just want to do something 30 times.

▼ F-string

- `f'{coeficiente}x^{exponente}'`: This is a formatted string literal. F-strings are a way to embed expressions inside string literals, using curly braces `{}`. The expressions will be replaced with their values when the string is created.

Syntax: `f'example etc {var}: continue example'`

▼ Slice strings

`[:]` from the beginning to ...

`[:]` from ... to the end

`[::-1]` goes to the end and iterates from the end to the left (beginning)

```
def run(text: str, target_word: str, replace_word: str) -> str:
    # TU CÓDIGO AQUÍ
    if target_word in text:
        # mtext = text.replace(target_word, replace_word)

        # haz lo mismo pero sin usar replace
```

```

    # Encuentra la posición inicial de la palabra objetivo
    start_pos = text.find(target_word)

    # Encuentra la posición final de la palabra objetivo
    end_pos = start_pos + len(target_word)

    # Forma el nuevo texto concatenando la parte antes del objetivo,
    # la palabra de reemplazo, y la parte después de la palabra objetivo
    mtext = text[:start_pos] + replace_word + text[end_pos:]

    return mtext

return mtext

if __name__ == "__main__":
    run("This is a beautiful night on the Atlantic", "beautiful", "terrible")

```

▼ Exceptions best practices

Here are some common best practices for exception handling in Python:

1. **Use built-in exceptions when appropriate:** Python has a number of built-in exceptions that can be used to indicate different types of errors. For example, you might raise a `ValueError` when a function receives an argument of the right type but inappropriate value.
2. **Create custom exceptions for specific error conditions:** If the built-in exceptions don't cover your needs, you can define your own by subclassing the `Exception` class or one of its subclasses.
3. **Don't use bare `except` clauses:** Catching all exceptions can hide errors and make debugging more difficult. Instead, catch specific exceptions that you know how to handle.
4. **Use the `finally` clause for cleanup code:** Code in a `finally` clause will be executed whether an exception was raised or not, making it a good place for cleanup code.
5. **Don't catch exceptions you can't handle:** If you don't know how to handle an exception, it's usually better to let it propagate up to a higher level where it can be handled appropriately.
6. **Provide useful error messages:** When raising an exception, include a descriptive message to help with debugging.
7. **Use exceptions for error conditions, not normal control flow:** Exceptions should be used to handle unexpected or error conditions. They should not be used for normal control flow in your program.
8. **Document the exceptions your function may raise:** If your function may raise an exception, document this in the function's docstring. This helps other developers understand how to use your function correctly.
9. **Don't suppress exceptions:** If you catch an exception, handle it in some way. Don't just pass and ignore it. This can lead to silent failures that are hard to debug.

▼ Decoradores

Los decoradores en Python son una forma de modificar o mejorar las funciones o los métodos existentes sin cambiar su código fuente. Un decorador es una función que toma otra función como argumento, añade alguna funcionalidad y luego devuelve la función original o una nueva función.

En Django, los decoradores se utilizan comúnmente para los siguientes propósitos:

1. Control de acceso: Los decoradores

como `login_required`, `permission_required` y `user_passes_test` se utilizan para restringir el acceso a ciertas vistas basándose en si el usuario está autenticado, si tiene ciertos permisos o si pasa ciertos tests.

2. Cache: Los decoradores como `cache_page` y `cache_control` se utilizan para almacenar en caché el resultado de una vista, lo que puede mejorar significativamente el rendimiento de tu aplicación.

3. Vistas genéricas: Los decoradores como `method_decorator` se utilizan para añadir decoradores a las vistas basadas en clases.

En tu código, los decoradores `@receiver(post_save, sender=User)` son ejemplos de decoradores de señales en Django. Las señales son una forma de notificación cuando ocurren ciertos eventos, como guardar un modelo. En este caso, los decoradores están diciendo:

"Cuando un objeto `User` se guarde, ejecuta la función `create_user_profile` o `save_user_profile`".

▼ Posibles bugs

Una lista de cosas inesperadas de las que Python no te avisa (no lanza mensajes de error cuando pasan*), y quizás el motivo del que pases tiempo de más debugueando.

▼ List comprehension de una tupla o string

No va a dar error (por lo que no te vas a enterar de por qué tu código no funciona como esperabas), y no va a generar la tupla.

Va a crear un generador.

```
myrange = (number for number in range(1, 6))
myrange: <generator object <genexpr> at 0x10b3732e0>
```

Pasa con strings también al ser inmutables.

▼ Functions return

Python funciones sin return, hacen un return implícito de `None`.

▼ Pandas

```
osas > gento_gento.py
import faker
import pandas as pd

fake = faker.Faker(['es-ES'])

names = [fake.name() for _ in range(30)]
desc = [fake.paragraph(2) for _ in range(30)]
desc = [fake.job() for _ in range(30)]
names_slug = [name.lower().replace(' ','-') for name in names]
names_slug = [name.lower().replace('á','a') for name in names_slug]
names_slug = [name.lower().replace('é','e') for name in names_slug]
names_slug = [name.lower().replace('í','i') for name in names_slug]
names_slug = [name.lower().replace('ó','o') for name in names_slug]
names_slug = [name.lower().replace('ú','u') for name in names_slug]
names_slug = [name.lower().replace('ñ','n') for name in names_slug]

df = pd.DataFrame(list(zip(names,names_slug,jobs,desc)), columns=['Nombre','Slug','Trabajo','Desc'])

print(pd.io.sql.get_schema(df, 'gento'))

for index, row in df.iterrows():
```