

#

# DESARROLLO WEB FULLSTACK

con Python y JavaScript

polotic  
misiones



# Lógica en la Web con...

polotic  
misiones



# Django - Logica

Es posible que deseemos cambiar lo que se muestra en nuestro sitio web en función de algunas condiciones. Por ejemplo, si visitamos el sitio [www.isitchristmas.com](http://www.isitchristmas.com), probablemente se encontrará con una página que se ve así:



# NO

# Django - Logica

¿Cuándo ese sitio web dice “SI”? Se imaginarán que lo hace cuando es 25 de Diciembre. Podemos hacer algo similar, en la que verificamos si es el día de Año Nuevo o no.

Para ello creamos una nueva aplicación Django:

1. Ejecutamos `python manage.py startapp anionuevo` en la terminal.
2. Editamos `settings.py` y agregamos “anionuevo” como una de nuestras `INSTALLED_APPS`
3. Editamos el archivo `urls.py` de nuestro proyecto e incluimos una ruta similar a la que creamos para la aplicación de saludo:

```
path('anionuevo/', include("anionuevo.urls"))
```

# Django - Logica

Creamos otro archivo `urls.py` dentro del directorio de nuestra nueva aplicación y lo actualizamos para incluir una ruta hacia la raíz (`index`) de la aplicación:

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

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

Cree una función de índice en `views.py`.

# Django - Logica



- Ahora que hemos configurado nuestra nueva aplicación, descubramos cómo comprobar si es el día de Año Nuevo o no.
- Para hacer esto, podemos importar el módulo de fecha y hora de Python. Para tener una idea de cómo funciona este módulo, podemos mirar la documentación y luego probarlo fuera de Django usando el intérprete de Python.
- El intérprete de Python es una herramienta que podemos usar para probar pequeños fragmentos de código Python. Para usar esto, ejecutas `python` en su terminal, y luego podrás escribir y ejecutar código Python dentro de tu terminal. Cuando hayas terminado de usar el intérprete, ejecutas `exit()` para salir.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.day
4
>>> now.month
6
>>> now.year
2020
>>> exit()
```



# Django - Logica



Podemos usar este conocimiento para construir una expresión booleana que se evaluará como True si y solo si hoy es el día de Año Nuevo: `now.day == 1 and now.month == 1`

Ahora que tenemos una expresión que podemos usar para evaluar si es el día de Año Nuevo o no, podemos actualizar nuestra función de índice en `views.py`:

```
def index(request):
    ahora = datetime.datetime.now()
    return render(request, "anionuevo/index.html", {
        "anionuevo": ahora.month == 1 and ahora.day == 1
    })
```

# Django - Logica

Ahora, creemos nuestra plantilla index.html.

Tendremos que volver a crear una nueva carpeta llamada templates, una carpeta dentro que la denominaré anionuevo y un archivo dentro llamado index.html.

En el código de la derecha, observarás que cuando deseamos incluir lógica en nuestros archivos HTML, usamos {% y %} como **etiquetas de apertura** y **cierre** alrededor de declaraciones lógicas. También ten en cuenta que el lenguaje de formato de Django requiere que incluyas una etiqueta final que indique que has terminado con el bloque if-else.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>¿Es año nuevo?</title>
  </head>
  <body>
    {% if anionuevo %}
      <h1>SI</h1>
    {% else %}
      <h1>NO</h1>
    {% endif %}
  </body>
</html>
```



# Django - Logica

Hay altas probabilidades de que estés leyendo y haciendo esto en un día que no es año nuevo. Entonces cada vez que cargas la página verás un NO. Sin embargo podemos poner un truco para probar que es lo que sucede cuando la condición es True:

```
def index(request):  
    ahora = datetime.datetime.now()  
    return render(request, "anionuevo/index.html", {  
        "anionuevo": True  
    })
```



# Trabajando con Estructuras o Secuencias de Datos en Django

# Django – Ej.: Listas de Tareas



Ahora hagamos un mini proyecto que incluya un listado de tareas a realizar.

Para ello creamos una nueva App como lo hicimos anteriormente.

1. Ejecutamos `python manage.py startapp tareas` en la terminal.
2. Editamos `settings.py`, agregando "tareas" como una de nuestras `INSTALLED_APPS`
3. Editamos el archivo `urls.py` de nuestro proyecto

Cree otro archivo `urls.py` dentro del directorio de nuestra nueva aplicación y lo actualizamos para incluir una ruta.

```
from django.shortcuts import render

tareas = ["foo", "bar", "baz"]

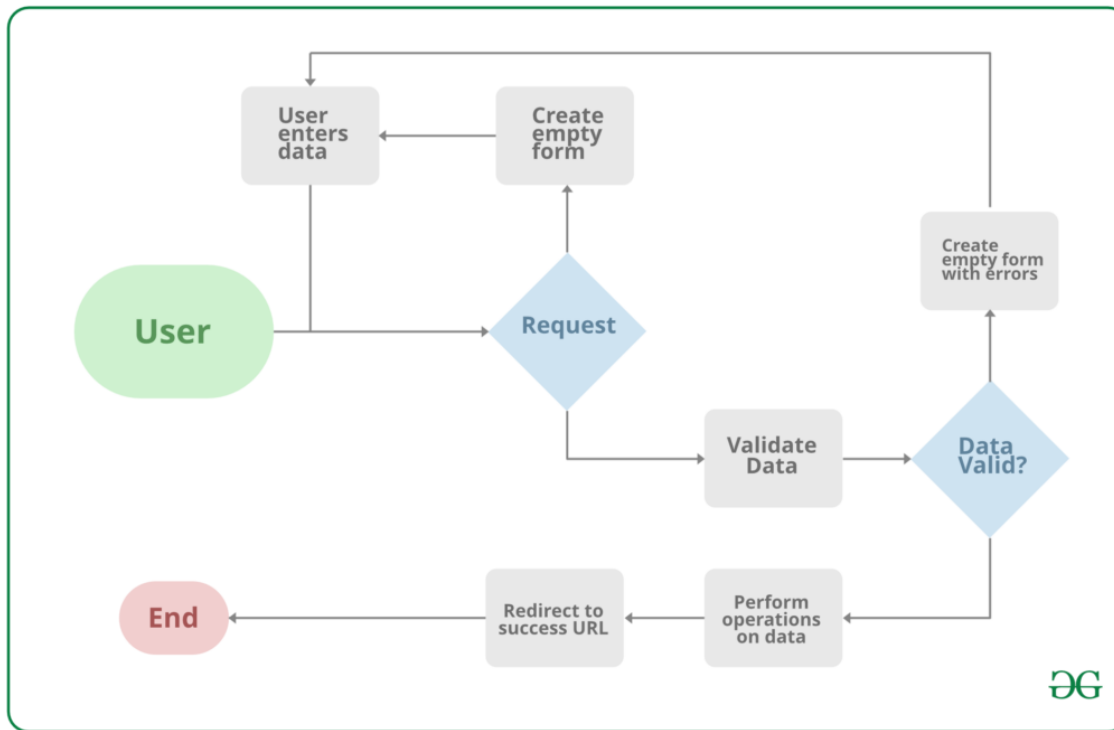
# Crear vistas aquí. def index(request):
    return render(request, "tareas/index.html", {
        "tareas": tareas
    })
```

# Django – Ej.: Listas de Tareas

Ahora, trabajemos en la creación de nuestro archivo HTML de plantilla, donde recorreremos la lista de tareas mediante un for:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>Tareas</title>
  </head>
  <body>
    <ul>
      {% for tarea in tareas %}
        <li>{{ tarea }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

# Formularios con Django





# Formularios con Django



Ahora que podemos ver nuestras tareas como listas de cosas, quizá querramos **añadir nuevas tareas**. Para hacer esto comenzamos utilizando formularios para actualizar los datos del sitio web.

Comencemos agregando otra función a views.py que generará una página con un formulario para agregar una nueva tarea:

```
# Agregar una nueva tarea: def agregar(request):  
    return render(request, "tareas/agregar.html")
```

A continuación, nos aseguramos de agregar otra ruta a urls.py:

```
path("agregar", views.agregar, name="agregar")
```



# Formularios con Django



Ahora, crearemos nuestro archivo agregar.html, que es bastante similar a index.html, excepto que en el cuerpo incluiremos un formulario en lugar de una lista:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Tareas</title>
</head>
<body>
    <h1>Agregar Tarea:</h1>
    <form action="">
        <input type="text", name="tarea">
        <input type="submit">
    </form>
</body>
</html>
```

# Formularios con Django

Sin embargo, lo que acabamos de hacer no es necesariamente el mejor diseño, ya que acabamos de repetir la mayor parte de ese HTML en dos archivos diferentes.

El lenguaje de plantillas de Django nos brinda una forma de eliminar este diseño deficiente: la herencia de plantillas. Esto nos permite crear un archivo `layout.html` que contendrá la estructura general de nuestra página:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Tareas</title>
</head>
<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

Utilizamos `{% ... %}` para denotar algún tipo de lógica no HTML



# Formularios con Django



Ahora, podemos alterar nuestros otros dos archivos HTML para que se vean así:

index.html

```
{% extends "tareas/layout.html" %}

{% block body %}
    <h1>Tareas:</h1>
    <ul>
        {% for unaTarea in tareas %}
            <li>{{ tarea }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

agregar.html

```
{% extends "tareas/layout.html" %}

{% block body %}
    <h1>Agregar Tarea:</h1>
    <form action="">
        <input type="text", name="tarea">
        <input type="submit">
    </form>
{% endblock %}
```



# Formularios con Django



Sin embargo, en lugar de codificar enlaces, ahora podemos usar la variable de nombre que asignamos a cada ruta en urls.py, y crear un enlace que se vea así:

```
<a href="{% url 'agregar' %}">Agregar una Nueva Tarea</a>
```

donde "agregar" es el nombre de esa ruta. Podemos hacer algo similar en nuestro agregar.html:

```
<a href="{% url 'index' %}">Ver Tareas</a>
```

Sin embargo, esto podría crear un problema, ya que tenemos algunas rutas denominadas index en nuestras diferentes aplicaciones. Podemos resolver esto yendo a cada uno de los archivos urls.py de nuestra aplicación y agregando una variable `app_name`, de modo que los archivos ahora se vean así:

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

app_name = "tareass"
urlpatterns = [
    path("", views.index, name="index"),
    path("agregar", views.agregar, name="agregar")
]
```

# Formularios con Django



Luego podemos cambiar nuestros enlaces de simplemente index y agregar a tareas: index y tareas: agregar

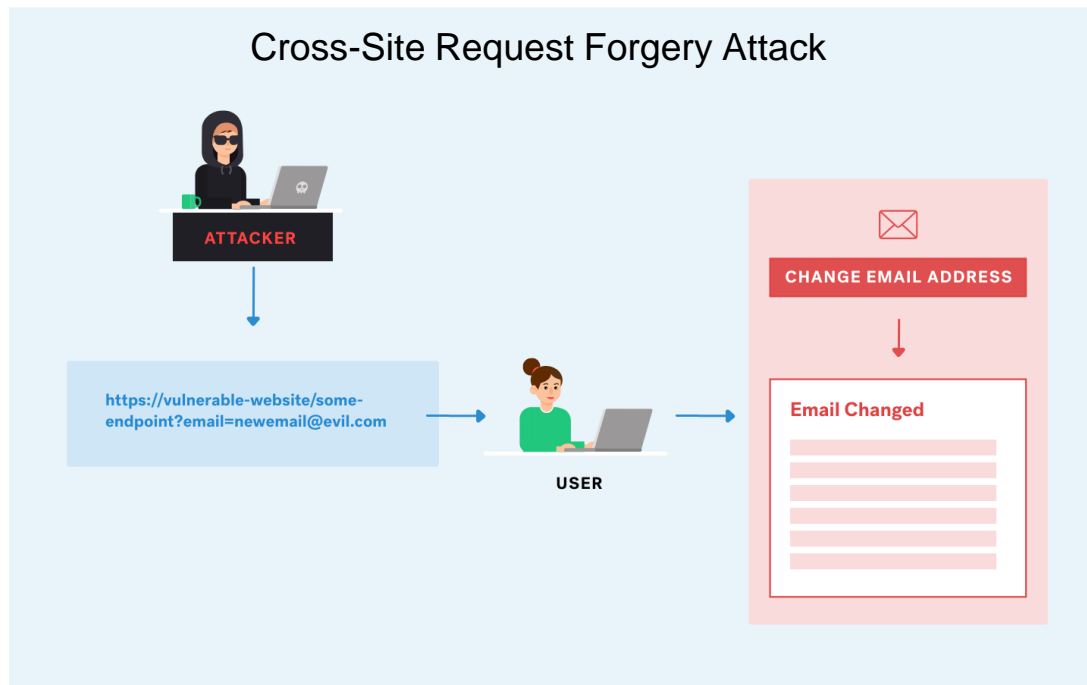
```
<a href="{% url 'tareas:index' %}">Ver Tareas</a>  
<a href="{% url 'tareas:agregar' %}">Agregar Nueva Tarea</a>
```

Ahora, trabajemos para asegurarnos de que el formulario realmente funcione cuando el usuario lo envía. Podemos hacer esto agregando una acción al formulario que hemos creado en agregar.htm

```
<form action="{% url 'tareas:agregar' %}" method="post">
```

Esto significa que una vez que se envíe el formulario, seremos redirigidos a la URL para agregar. Aquí hemos especificado que usaremos un método de publicación en lugar de un método de obtención, que normalmente es lo que usamos cada vez que un formulario podría alterar el estado de esa página web.

# Seguridad en Django





# Seguridad en Django

- Necesitamos agregar un poco más de seguridad para proteger nuestros formularios de los ataques, y lo hacemos con una funcionalidad que ofrece Django en la cual se implementa un token para evitar el ataque de falsificación de solicitudes entre sitios (CSRF).
- Este es un ataque en el que un usuario malintencionado intenta enviar una solicitud a su servidor desde otro lugar que no sea su sitio. Esto podría ser un gran problema para algunos sitios web. Digamos, por ejemplo, que un sitio web bancario tiene un formulario para que un usuario transfiera dinero a otro. ¡Sería catastrófico si alguien pudiera enviar una transferencia desde fuera del sitio web del banco!



# Seguridad en Django - CSRF

- Para resolver este problema, cuando Django envía una respuesta que representa una plantilla, también proporciona un token CSRF que es único con cada nueva sesión en el sitio.
- Luego, cuando se envía una solicitud, Django verifica para asegurarse de que el token CSRF asociado con la solicitud coincida con uno que haya proporcionado recientemente. Por lo tanto, si un usuario malintencionado en otro sitio intentara enviar una solicitud, sería bloqueado debido a un token CSRF no válido. Esta validación CSRF está integrada en el marco de [Django Middleware](#), que puede intervenir en el procesamiento de solicitudes y respuestas de una aplicación Django. No entraremos en más detalles sobre Middleware en este curso, pero puedes ver la documentacion.

# Seguridad en Django - CSRF

```
<form action="{% url 'tareas:agregar' %}" method="post">
    {% csrf_token %}
    <input type="text", name="tarea">
    <input type="submit">
</form>
```

- Esta línea agrega un campo de entrada oculto con el token CSRF proporcionado por Django, de modo que cuando recargamos la página, parece que nada ha cambiado. Sin embargo, si inspeccionamos el elemento, notaremos que se ha agregado un nuevo campo de entrada.



# Django Forms



Si bien podemos crear formularios escribiendo HTML puro como acabamos de hacer, Django proporciona una forma aún más fácil de generar formularios automáticamente: Django Forms.

Para utilizar ésta funcionalidad, agregaremos lo siguiente en la parte superior de `views.py` para importar el módulo `forms`:

```
from django import forms
```

Ahora, podemos crear un nuevo formulario dentro de `views.py` creando una clase Python llamada `FormNuevaTarea`:

```
class FormNuevaTarea(forms.Form):  
    tarea = forms.CharField(label="Nueva Tarea")
```



# Django Forms

```
class FormNuevaTarea(forms.Form):  
    tarea = forms.CharField(label="Nueva Tarea")
```

Mi clase formulario  
heredará de la clase  
Form para poder  
implementarla

El input se reduce  
aquí a una variable  
de Python

Éste es el tipo del  
input, en éste caso  
un campo de  
caracteres. [Hay  
muchos otros  
campos para elegir.](#)

Uno de los  
parámetros que  
puedo modificar de  
éste input es el  
label que se  
mostrará al usuario

# Django Forms

Ahora que hemos creado una clase `FormNuevaTarea`, podemos incluirla en el contexto mientras renderizamos la página para agregar:

```
# Agregar nueva tarea: def agregar(request):  
    return render(request, "tareas/agregar.html", {  
        "form": FormNuevaTarea()  
    })
```

Ahora, dentro de `agregar.html`, podemos reemplazar nuestro campo de entrada con el formulario que acabamos de crear:

```
{% extends "tareas/layout.html" %}  
  
{% block body %}  
    <h1>Agregar Tarea:</h1>  
    <form action="{% url 'tareas:agregar' %}" method="post">  
        {% csrf_token %}  
        {{ form }}  
        <input type="submit">  
    </form>  
    <a href="{% url 'tareas:index' %}">Ver Tareas</a>  
{% endblock %}
```

# Django Forms

Hay varias ventajas de usar el módulo de forms en lugar de escribir manualmente un formulario HTML:

- Si queremos agregar nuevos campos al formulario, simplemente podemos agregarlos en `views.py` sin escribir HTML adicional.
- Django realiza automáticamente la [validación del lado del cliente](#) o la validación local en la máquina del usuario. lo que significa que no permitirá que un usuario envíe el formulario si está incompleto.
- Django proporciona una [validación simple del lado del servidor](#), o validación que ocurre una vez que los datos del formulario han llegado al servidor.
- En la próxima clase, comenzaremos a usar **modelos** para almacenar información, y Django simplifica la creación de un formulario basado en un modelo.



# Django Forms



Ahora que tenemos un formulario configurado, trabajemos en lo que sucede cuando un usuario hace clic en el botón Enviar/Submit.

Cuando un usuario navega a la página de agregar haciendo clic en un enlace o escribiendo la URL, envía una solicitud GET al servidor, que ya hemos manejado en nuestra función de agregar. Sin embargo, cuando un usuario envía un formulario, envía una solicitud POST al servidor, que por el momento no se maneja en la función de agregar.

Podemos manejar un método POST agregando una condición basada en el argumento de solicitud que toma nuestra función. Los comentarios en el código a continuación explican el propósito de cada línea:

# Django Forms

```
# Agregar una nueva Tarea: def agregar(request):
    # Chequear si el metodo es POST if request.method == "POST":
        # Tomar los datos que el usuario ha enviado y guardarlo como un formulario
        form = FormNuevaTarea(request.POST)
        # Chequear si el dato del formulario es valido (lado servidor) if form.is_valid():
            # Aislar la tarea obteniendo una version 'limpia' de los datos tarea = form.cleaned_data["tarea"]
            # Agregar la nueva tarea a nuestra lista de tareas tareas.append(tarea)
            # Redireccionar al usuario a la lista de tareas return HttpResponseRedirect(reverse("tareas:index"))
        else:
            # Si el formulario es invalido, volver a renderizar la pagina con la informacion existente.
            return render(request, "tareas/agregar.html", {
                "form": form
            })
    return render(request, "tareas/agregar.html", {
        "form": FormNuevaTarea()
    })
```



# Django Forms



Una nota rápida: para redirigir al usuario después de un envío exitoso, necesitamos algunas importaciones más:

```
from django.urls import reverse  
from django.http import HttpResponseRedirect
```



# Manejo de Sesiones



En este punto, hemos creado con éxito una aplicación que nos permite agregar tareas a una lista en crecimiento.

Sin embargo, puede ser un problema que almacenemos estas tareas como una variable global, ya que significa que todos los usuarios que visitan la página ven exactamente la misma lista.

Para solucionar este problema vamos a emplear una herramienta conocida como sesiones.

Las sesiones son una forma de almacenar datos únicos en el lado del servidor para cada nueva visita a un sitio web.

Para usar sesiones en nuestra aplicación, primero borraremos nuestra variable de tareas globales, luego modificaremos nuestra función `index` y finalmente nos aseguraremos de que en cualquier otro lugar hayamos usado la variable `tareas`, la reemplazamos con `request.session["tareas"]`

# Manejo de Sesiones

```
def index(request):
    # Chequear si ya existe una clave "tareas" en nuestra sesion
    if "tareas" not in request.session:
        # Si no, crear una nueva lista request.session["tareas"] = []
    return render(request, "tareas/index.html", {
        "tareas": request.session["tareas"]
    })

# Agregar nueva tarea: def add(request):
    if request.method == "POST":
        # Tomar los datos que el usuario envia y guardarlos como form form = FormNuevaTarea(request.POST)
        # Chequear si los datos del formulario es valido (lado del servidor) if form.is_valid():
            # Aislar la tarea de la version limpia del form tarea = form.cleaned_data["tarea"]
            # Agregar una nueva tarea a nuestra lista de tareas request.session["tareas"] += [task]
            # Redireccionar usuario a la lista de tareas return HttpResponseRedirect(reverse("tareas:index"))
        else:
            # Si el formulario es invalido, volver a renderizar la pagina con la informacion existente. return render(request, "tareas/agregar.html", {
                "form": form
            })
    return render(request, "tareas/agregar.html", {
        "form": FormNuevaTarea ()
    })
```





# Manejo de Sesiones



Finalmente, antes de que Django pueda almacenar estos datos, debemos ejecutar

```
python manage.py migrate
```

 en el terminal.

La semana que viene hablaremos más sobre qué es una migración, pero por ahora solo debes saber que el comando anterior nos permite almacenar sesiones.

# Modelos

django





# Modelos en Django



De la misma manera que pensábamos los Objetos en POO con Python, aquí también pensamos los Modelos como los elementos que conforman la lógica de negocios de nuestra aplicación y representan la realidad.

Los [Models](#) son un nivel de [abstracción](#) sobre la capa de bases de datos que usan clases y objetos de Python en lugar de consultas directas a la BD.

# Modelos en Django



Comencemos a usar modelos creando un proyecto de Django para (por ejemplo) una aerolínea y sus vuelos. Para ello necesitamos crear una aplicación dentro de un proyecto.

```
django-admin startproject aerolinea
cd aerolinea
python manage.py startapp vuelos
```

Ahora tendremos que pasar por el proceso de agregar una aplicación como de costumbre:

- Agrega vuelos a la lista `INSTALLED_APPS` en `settings.py`
- Agrega una ruta para vuelos en `urls.py`:

```
path("vuelos/", include("vuelos.urls")),
```

- Crea un archivo `urls.py` dentro de la aplicación de vuelos. Y completa con los imports y las listas de url estándar de `urls.py`

# Modelos en Django



Ahora, en lugar de crear rutas reales y comenzar con `views.py`, crearemos algunos modelos en el archivo `models.py`. En este archivo, describiremos qué datos queremos almacenar en nuestra aplicación. Luego, Django determinará la sintaxis de Base de Datos necesaria para almacenar la información en cada uno de nuestros modelos. Echemos un vistazo a cómo se vería un modelo para un solo vuelo:

```
class Vuelo(models.Model):  
    origen = models.CharField(max_length=64)  
    destino = models.CharField(max_length=64)  
    duracion = models.IntegerField()
```

# Modelos en Django

```
class Vuelo(models.Model):  
    origen = models.CharField(max_length=64)  
    destino = models.CharField(max_length=64)  
    duracion = models.IntegerField()
```

Echemos un vistazo a lo que sucede en esta definición de modelo:

- ❖ En la primera línea, creamos un nuevo modelo que amplía la clase Model de Django.
- ❖ A continuación, agregamos campos para origen, destino y duración. Los dos primeros son campos de [caracteres](#), lo que significa que almacenan cadenas (Strings), y el tercero es un campo [entero](#). Estas son solo dos de las muchas [clases de Django Field integradas](#).
- ❖ Especificamos longitudes máximas de 64 elementos para los dos campos de caracteres. Puedes consultar las especificaciones disponibles para un campo determinado consultando la [documentación](#).



# Migrations



Ahora, aunque hemos creado un modelo, todavía no tenemos una base de datos para almacenar esta información. para crear una base de datos a partir de nuestros modelos, navegamos al directorio principal de nuestro proyecto y ejecutamos el comando:

```
python manage.py makemigrations
```

Este comando crea unos archivos de Python que crearán o editarán nuestra base de datos para poder almacenar lo que tenemos en nuestros modelos. Debería obtener un resultado similar al que se muestra a continuación y, si navega a su directorio de migraciones, notará que se creó un nuevo archivo para nosotros.

A continuación, para aplicar estas migraciones a nuestra base de datos, ejecutamos el comando

```
python manage.py migrate
```

Ahora, verás que se han aplicado algunas migraciones predeterminadas junto con las propias, y también notarás que ahora tenemos un archivo llamado `db.sqlite3` en el directorio de nuestro proyecto.



# Migrations



`db.sqlite3` es el archivo de Base de Datos SQL por defecto de Django, denominado **SQLite**.

En Django podemos usar los sistemas de Bases de Datos que querramos (tanto SQL o NoSQL).

*Esto lo veremos la clase que viene*



# Shell

Ahora, para comenzar a trabajar agregando información y manipulando esta base de datos, podemos ingresar al shell de Django donde podemos ejecutar comandos de Python dentro de nuestro proyecto.

Para acceder al Shell ejecutamos el siguiente comando:

```
python manage.py shell
```

SHELL

```
# Importar el modelo Vuelo
In [1]: from vuelos.models import Vuelo

# Crear un nuevo Vuelo
In [2]: v = Vuelo(origen="Buenos Aires", destino="Iguazu", duracion=2)

# Persistir ese vuelo en la base de datos
In [3]: v.save()

# Consultar por todos los vuelos guardados en la Base de Datos
In [4]: Vuelo.objects.all()
Out [4]: <QuerySet [<Vuelo: Vuelo object (1)>]>
```

# Shell

Cuando consultamos los vuelos podemos ver que se retorna un objeto <Vuelo: Vuelo object (1)>

Este no es un nombre muy informativo, pero podemos solucionarlo. Dentro de models.py, definiremos una función `__str__` que proporciona instrucciones sobre cómo convertir un objeto Vuelo en una cadena:

```
class Vuelo(models.Model):
    origen = models.CharField(max_length=64)
    destino = models.CharField(max_length=64)
    duracion = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origen} a {self.destino}"
```

# Shell

## SHELL

```
# Crea una variable llamada vuelos para almacenar los resultados de una consulta.  
In [7]: vuelos = Vuelo.objects.all()
```

```
# Visualización de todos los vuelos
```

```
In [8]: vuelos
```

```
Out[8]: <QuerySet [<Vuelo: 1: Buenos Aires a Iguazu>]>
```

```
# Aislando solo el primer vuelo
```

```
In [9]: un_vuelo = vuelos.first()
```

```
# Impresión de información de vuelo
```

```
In [10]: un_vuelo
```

```
Out[10]: <Vuelo: 1: Buenos Aires a Iguazu>
```

```
# Mostrar identificación de vuelo
```

```
In [11]: un_vuelo.id
```

```
Out[11]: 1
```

```
# Mostrar el origen del vuelo
```

```
In [12]: un_vuelo.origen
```

```
Out[12]: 'Buenos Aires'
```

```
# Mostrar destino de vuelo
```

```
In [13]: un_vuelo.destino
```

```
Out[13]: 'Iguazu'
```

```
# Mostrar la duración del vuelo
```

```
In [14]: un_vuelo.duracion
```

```
Out[14]: 2
```



# Mostrar los Modelos en la Web



Ahora podemos comenzar a construir una aplicación en torno a este proceso de usar modelos para interactuar con una base de datos. Comencemos por crear una ruta `index` para nuestra lista de **vuelos**. Dentro de `urls.py`:

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

# Mostrar los Modelos en la Web



Dentro de `views.py`:

```
from django.shortcuts import render
from .models import Vuelo

# Create your views here.

def index(request):
    return render(request, "vuelos/index.html", {
        "vuelos": Vuelo.objects.all()
    })
```

# Mostrar los Modelos en la Web



Dentro de los templates vamos a crear un template padre que se llame `layout.html`:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>Vuelos</title>
  </head>
  <body>
    {% block body %}
    {% endblock %}
  </body>
</html>
```

# Mostrar los Modelos en la Web



Dentro de los templates vamos a crear un template que se llame index.html:

```
{% extends " vuelos/layout.html" %}

{% block body %}
    <h1>Vuelos:</h1>
    <ul>
        {% for un_vuelo in vuelos %}
            <li>Vuelo {{vuelo.id }}: {{vuelo.origen }} a {{vuelo.destino }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```



# Django Admin



Dado que es tan común que los desarrolladores tengan que crear nuevos objetos como lo hemos estado haciendo en el shell, Django viene con una interfaz de administración predeterminada que nos permite hacer esto más fácilmente. Para comenzar a utilizar esta herramienta, primero debemos crear un usuario administrativo con el siguiente comando:

```
python manage.py createsuperuser
```

Tenemos que proveer un Username (nombre de usuario), un Email Address (dirección de correo electrónico), un Password (y repetirlo nuevamente para confirmarlo) para así poder acceder con esas credenciales.



# Django Admin



Ahora, debemos agregar nuestros modelos a la aplicación de administración ingresando al archivo `admin.py` dentro de nuestra aplicación e importando y registrando nuestros modelos.

Esto le dice a Django a qué modelos nos gustaría tener acceso en la aplicación de administración.

```
from django.contrib import admin
from .models import Vuelo

# Register your models here.
admin.site.register(Vuelo)
```

# Django Admin



Ahora, cuando visitamos nuestro sitio y agregamos `/admin` a la URL, podemos iniciar sesión en una página que se ve así

A screenshot of the Django administration login interface. At the top, a dark blue header bar contains the text 'Django administration' in white. Below this, the form is set against a light gray background. It includes a 'Username:' label followed by a text input field containing 'user\_a'. Below that is a 'Password:' label followed by a password input field with a single dot visible. At the bottom center of the form is a blue 'Log in' button.

Django administration

Username:

Password:

Log in

# Django Admin



Después de iniciar sesión, accederás a una página como la que se muestra a continuación, donde podrás crear, editar y eliminar objetos almacenados en la base de datos.


Django administration

WELCOME, **USER\_A**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)


Site administration

AUTHENTICATION AND AUTHORIZATION

Groups


+ Add    Change

Users

+ Add    Change

VUELOS

Vuelos

+ Add    Change

Recent actions

My actions

None available

# Django Admin



En ésta página ya tendrás los formularios para cargar los modelos (Add) o para editarlos (Change). Todas las acciones se registran en un historial que verás en la barra lateral derecha.


Django administration

WELCOME, **USER\_A**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)


Site administration

AUTHENTICATION AND AUTHORIZATION

Groups


+ Add    Change

Users

+ Add    Change

VUELOS

Vuelos

+ Add    Change

Recent actions

My actions

None available



# Modelos: ¿Qué sigue...?

Aprendimos a crear modelos personalizados, pero aún no hemos aprendido a vincularlos entre sí. Esto será tema de la clase siguiente donde veremos:

- La estructura y el funcionamiento de una base de datos SQL y como Django lo refleja
- Como definir el acceso a una Base de Datos en Django y como ocultar información sensible con Python Decouple
- Como enlazar modelos entre sí mediante claves
- Como enlazar modelos entre sí mediante relaciones de muchos a muchos
- Como autenticar usuarios para acceder al sistema y gestionar que pueden ver



# Trabajemos Juntos



Llegó la hora del desafío:

- Crea un proyecto y la aplicación en Django para el Trabajo Final. Comienza construyendo los Modelos primitivos que iras a necesitar mas adelante y así ir completando la estructura.



#

¡HASTA LA  
*próxima!*

[www.polotic.misiones.gob.ar](http://www.polotic.misiones.gob.ar)

[f](#) [@](#) [v](#) [d](#) /poloticmisiones

polotic  
misiones



Gobierno  
de Misiones

