# Django y su integración con bases de datos

## El ORM de Django

{desafío} latam_

***Describir las características fundamentales de la integración del framework Django con bases de datos.***

***Reconocer las aplicaciones preinstaladas con el motor Django distinguiendo su utilidad como apoyo al desarrollo.***

- Unidad 1:
  Django y su integración con bases de datos
- Unidad 2:
  Modelos de datos y el ORM de Django
- Unidad 3:
  Proyecto

***Te encuentras aquí***

{desafío} latam_

# ¿Qué aprenderás en esta sesión?

- *Reconoce las aplicaciones preinstaladas de Django y su utilidad como apoyo al desarrollo.*

- *Inspecciona el modelo Django para aplicaciones preinstaladas*

{desafío} latam_

# Recordemos:

# ¿Que es el ORM de Django?

# /* Los modelos */

# Los modelos

Según la documentación oficial de Django, un modelo es fuente única y definitiva de información sobre tus datos. Este contiene los campos esenciales y comportamientos de los datos que estás almacenando, generalmente cada modelo es mapeado a una sola base de datos.

- Cada modelo es una clase de python que hereda de **django.db.models.Model**
- Cada atributo del modelo representa un campo de la base de datos.
- Los modelos son definidos usualmente en el archivo models.py generado automáticamente por el manager de django al crear una aplicación nueva.

# Los modelos

```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Muestra la definición de un modelo Persona, que posee dos campos, llamados first_name y last_name, ambos del tipo Char de 30 caracteres de largo.

**Fuente**: djangoproject.com

# Los modelos

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

Muestra el equivalente en una sentencia de SQL del modelo de la imagen anterior.

**Fuente**: djangoproject.com

# /* Tipos de variables del ORM */

# Tipos de variables del ORM

Algunos de los tipos de datos más comunes para asignar a las propiedades de nuestro modelo son:

- **CharField**: Que es usado para definir cadenas de texto de largo fijo. Estas son de un tamaño pequeño a medio. Se puede definir el largo máximo de estas con el parámetro max_length.

- **TextField**: Es usado para cadenas de texto de largo arbitrario. Se puede especificar un largo máximo del campo, pero, por lo general esto se usa cuando el campo es desplegado en un formulario (por ejemplo, escriba su experiencia profesional, máximo 400 caracteres.), el largo definido en este caso aplica solamente a la aplicación y no se define en la base de datos.

# Tipos de variables del ORM

- **IntegerField**: Es un campo para almacenar números enteros.

- **DateField** y **DateTimeField**: Son usados para almacenar o representar información de fechas y fechas/tiempo. (Python datetime.date y datetime.datetime). También, estos campos pueden adicionalmente declarar el parámetro *auto_now=True* para actualizar la fecha actual en cada actualización del campo, o *auto_now_add=True* para guardar la fecha solamente cuando se crea el modelo y default para setear una fecha por defecto que puede ser sobreescrita por el usuario.

- **EmailField**: Es utilizada para almacenar direcciones de correo.

# Tipos de variables del ORM

- **AutoField**: Es un tipo especial de IntegerField, que se incrementa automáticamente. Una clave primaria de este tipo es automáticamente incrementada en tu modelo si no lo haces explícitamente.

- **ForeignKey**: Es un tipo de relación uno a muchos hacia otro modelo y es el equivalente a las claves foráneas en SQL. (ej. Un auto tiene un fabricante, pero un fabricante construye muchos autos).
  En las claves foráneas, podemos utilizar el argumento on_delete=models.CASCADE, esté argumentos nos permite, que cuando un objeto referenciado, sea eliminado, también se eliminan los elementos que tienen referencias hacia el.

- **ManyToManyField**: Es utilizada para generar relaciones muchos a muchos (ej. un libro puede tener muchos géneros, y cada género puede tener muchos libros.), estas relaciones, necesitan de una tabla intermedia en la base de datos, que es generada por el ORM.

# Tipos de variables del ORM

## *Ejemplo*

```
from django.db import models

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

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

En esta imagen podemos ver la creación de dos modelos Musician y Álbum, donde Álbum contiene una clave foránea (ForeignKey) hacia el modelo Musician.

**Fuente**: djangoproject.com

# /* Operaciones sobre objetos */

# Operaciones sobre objetos

## *Guardar*

En esta imagen podemos ver un ejemplo de guardar un objeto en la base de datos.
Primero creamos una instancia del objeto Blog, con sus correspondientes campos en el constructor.

Luego, se revisa b2.id, ya que es un campo autonumérico, vemos que no genera el valor antes de ser guardado. Id es generado automáticamente por el ORM si nosotros no definimos explícitamente un campo con el parámetro primary_key=True. Este registro genera el valor, solo cuando es guardado el registro en la base de datos.

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b2 doesn't have an ID yet.
```

# Operaciones sobre objetos

## *Guardar*

Luego utilizamos el método save y si no obtenemos ningún error, los datos quedan persistiendo en la base de datos. Podemos ver que fue así, al leer el valor de id, y nos debería indicar el correlativo generado.

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b2 doesn't have an ID yet.
>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
```

# Operaciones sobre objetos

## *Recuperando información*

Podemos recuperar información mediante una serie de metodos presentes en el modelo.

Por ejemplo, en esta imagen, usamos el method **all()**, que sería el equivalente a hacer **SELECT * FROM Entry** en sql.

En la imagen tenemos:

- **Entry**: el nombre del Modelo
- **objects**: identifica que se llamarán a los objetos del Modelo.
- **all()**: La manera en que se llamarán (en este caso, TODOS).

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

{desafío} latam_

# Operaciones sobre objetos

## *Recuperando información*

Cuando realizamos una recuperación de datos con este método, los datos son devueltos en un objeto tipo QuerySet. Un QuerySet es un objeto del ORM que es iterable que se comporta como una lista, conteniendo objetos del tipo Entry.

El QuerySet lo podemos recorrer iterando con el ciclo for, como muestra la imagen

```
for e in Entry.objects.all():
    print(e.headline)
```

# Operaciones sobre objetos

## *Recuperando información*

Tenemos otros métodos útiles para recuperar información desde los modelos, como son:

- **Filter**: Nos permite obtener valores filtrando por algún campo.

```
r1 = Entry.objects.filter(title=valor) # obtenemos un queryset con las
coincidencias encontradas.
# Equivalente en SQL: SELECT * FROM Entry WHERE title='valor'

r2 = Entry.objects.filter(title=valor).first() #obtenemos la primera
coincidencia, en un objeto del tipo Entry (en este caso)
# Equivalente en SQL: SELECT * FROM Entry WHERE title='valor' LIMIT 1

r3 = Entry.objects.get(id=valor) #obtenemos un único registro si hay
coincidencias. Esto se utiliza para claves que sabemos de antemano son
únicas.
#Equivalente en SQL: SELECT * FROM Entry where id = valor
```

# Operaciones sobre objetos

## *Recuperando información*

Ordenando la información que se obtendrá.

```
Entry.objects.order_by('blog').distinct('blog') # Tenemos order_by
para ordenar el QuerySet que nos retornará la consulta y distinct para
evitar duplicar filas.
#SQL: SELECT distinct blog from Entry order by blog
```

# Operaciones sobre objetos

## *Recuperando información*

Actualizando un la información de un objeto:

Aquí mostraremos una opción de actualización, recuperando un objeto y modificandolo.

```
e1 = Entry.objects.get(id=100) # obtenemos el registro con id 100
e1.title = "Nuevo título"  # Actualizamos el título existente.
e1.save() # se guarda el registro con los cambios realizados.
#SQL: UPDATE Entry set title=’Nuevo Titulo’ WHERE id=100
```

# Operaciones sobre objetos

## *Recuperando información*

Cuando queremos recuperar elementos de una relación, podemos hacerlo de dos formas, para este ejemplo utilizaremos la relación de la imagen de la diapositiva 13.

```
artist = Musician(first_name="Robert", last_name="Smith", instrument="Guitar")
artist.save()

#SQL: INSERT INTO Musician(first_name, last_name, instrument)
#     VALUES('Robert', 'Smith', 'Guitar')

album1 = Album(artist=artist, name="Album1", release_date='2020-01-01', stars=5)
album2 = Album(artist=artist, name="Album2", release_date='2021-01-01', stars=1)
album1.save()
album2.save()
```

# Operaciones sobre objetos

## *Recuperando información*

La primera opción es recuperar un álbum y tener todos sus datos.

```
album = Album.objects.get(name="Album 1")
print(album.artist.first_name)
```

# Operaciones sobre objetos

## *Recuperando información*

La segunda opción es hacer una consulta reversa desde Musician, para esto utilizamos el método **_set()** después del nombre del modelo en minúscula.

Nota: en estos casos estamos seguros de no tener otro Robert u otros Álbumes en la base de datos, por eso utilizamos get, que devuelve un único resultado, si hubieran más, nos lanzaría una excepción. Para esos casos utilizar filter(campos…).first()

```
artist = Musician.objects.get("first_name="Robert")
albumes = artist.album_set.all()
for album in albumes:
     print(album.name)
```

# /* Migraciones */

# Migraciones

Como vimos anteriormente, las migraciones nos permiten reflejar los cambios en los modelos hacia la base de datos (estructura, nuevos campos en algún objeto, etc…).

Estas se componen principalmente de dos comandos:

- makemigrations
- migrate

{desafío} latam_

# Migraciones
## *makemigrations*

- **makemigrations**:

  Este comando realiza una búsqueda de cambios en los objetos que componen el proyecto, tanto creación, actualización y eliminación de modelos.

  Los cambios son almacenados en un directorio llamado migrations dentro de cada aplicación, y son numerados para mantener una relación, cada migración guarda referencia a la migración anterior para así tener un historial de cambios completo y poder aplicar todos los cambios en caso de configurar la aplicación a una nueva base de datos.

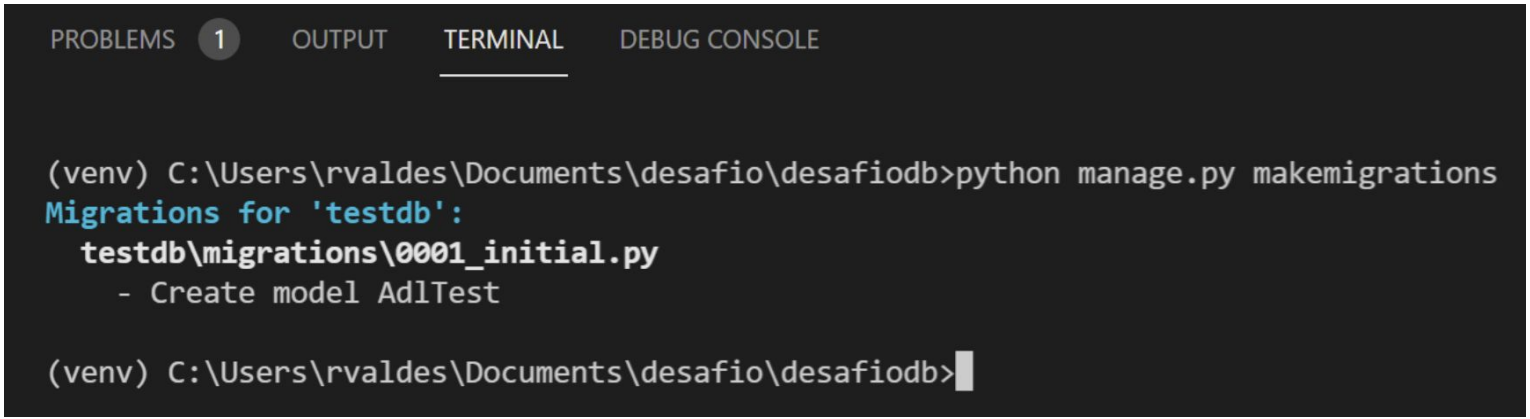
# Migraciones
## *migrate*

- **migrate**:

  Este comando se ejecuta para aplicar las migraciones creadas con makemigrations a la base de datos, también nos permite volver a una migración anterior.
  (python manage.py migrate nombre_app nombre_migracion_anterior)

# Migraciones

Aquí vemos un ejemplo de la ejecución del comando:
**python manage.py makemigrations** en un proyecto con un nuevo modelo.



# Migraciones

Aquí vemos la aplicación de las migraciones sobre un proyecto nuevo.

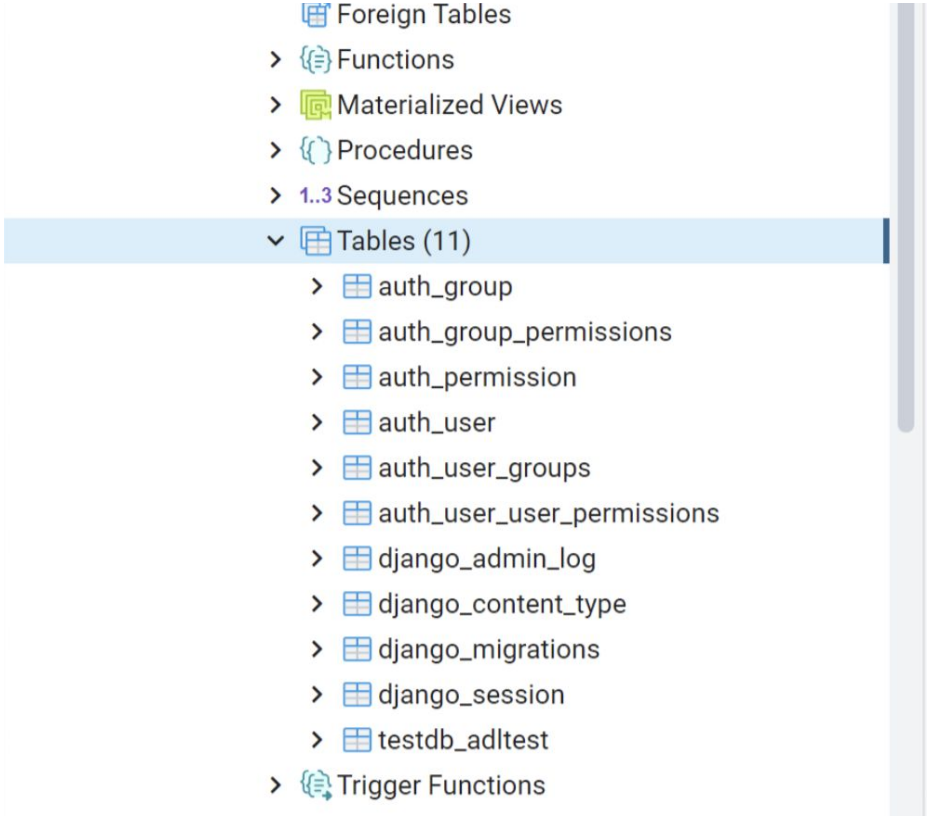
Cuando recién creamos un proyecto django, deben generarse una serie de tablas en la base de datos que utilizará el sistema en sí, más los modelos que hayamos creado nosotros.

```
(venv) C:\Users\rvaldes\Documents\desafio\desafiodb>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, testdb
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying testdb.0001_initial... OK

(venv) C:\Users\rvaldes\Documents\desafio\desafiodb>
```

# Migraciones

Luego de migrar los cambios, podemos ver estos reflejados en la base de datos.



# /* Aplicaciones Pre-Instaladas de Django */

# Aplicaciones Pre-Instaladas de Django

Django por defecto trae una serie de aplicaciones pre-instaladas que nos entregan gran parte de la funcionalidad del mismo. Podemos ver estas aplicaciones en el archivo settings.py

```
# Application definition

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

# Aplicaciones Pre-Instaladas de Django

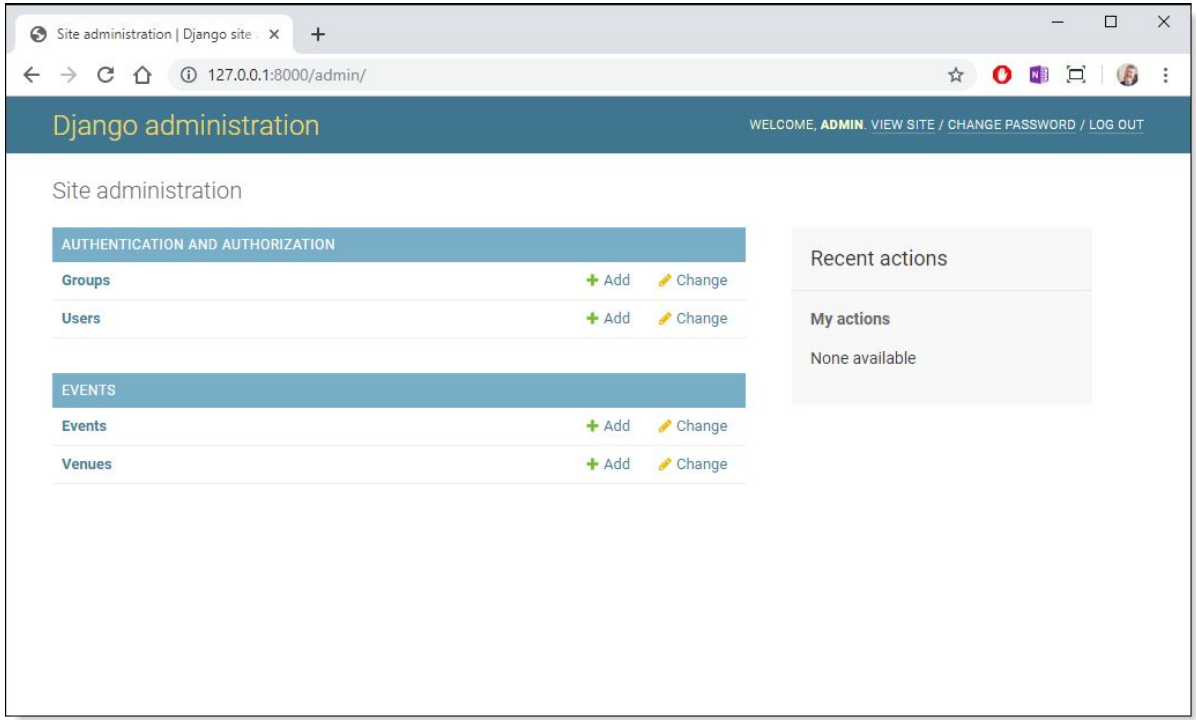
## *django.contrib.admin*

Este es otro componente que hace a django tremendamente competitivo frente a otros frameworks, ya que gracias a este administrador prefabricado que trae, cada vez que agreguemos un nuevo modelo al App, podremos listar, agregar, editar y eliminar registros del mismo a través de interfaz gráfica para realizar pruebas tempranas del mismo, solo agregando unas líneas en la configuración del admin.

Para acceder a este administrador, cuando ejecutamos el proyecto, django nos disponibiliza la siguiente url: http://127.0.0.1:8000/admin/.

# Aplicaciones Pre-Instaladas de Django

## *Imagen del administrador*




**Fuente**: djangobook.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.admin*

En la siguiente imagen, podemos ver la forma de agregar nuestros modelos al admin.
En este caso, importamos el objeto Admin, y nuestro modelo, luego registramos el modelo. Esto lo realizamos en el archivo **admin.py** que se encuentra en la raíz del directorio de cada aplicación.

```
from django.contrib import admin
from myproject.myapp.models import Author

admin.site.register(Author)
```

**Fuente**: docs.djangoproject.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.auth*

Esta aplicación maneja todo lo relacionado con la seguridad de un sitio generado con Django.

Tiene un modelo de usuario genérico llamado User con los campos más usuales, el cual se puede extender de ser necesario.

También nos entrega herramientas de autenticación, manejo de passwords, sesiones de usuario, grupos y permisos.

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.auth*

Podemos ver un ejemplo de la utilización de los recursos de esta aplicación.

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

django creación de usuarios

**Fuente**: docs.djangoproject.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.auth*

Podemos ver otro ejemplo de la utilización de los recursos de esta aplicación.

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # A backend authenticated the credentials
else:
    # No backend authenticated the credentials
```

django autenticación de usuarios

**Fuente**: docs.djangoproject.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.contenttypes*

Esta aplicación lleva registro de todos los modelos instalados o creados en tu proyecto django. Provee una interfaz genérica de alto nivel para trabajar con estos modelos.

Cada instancia de ContentType tiene métodos que permiten obtener una instancia del modelo representado en ContentTypes o recuperar objetos de ese modelo.

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.contenttypes*

En la imagen, tenemos un ejemplo donde recuperamos el modelo user de la aplicación “auth” a partir del registro contenido en ContentTypes. Entonces, para utilizarlo llamamos el método model_class() y esto nos entrega la clase del modelo.

```
>>> from django.contrib.contenttypes.models import ContentType
>>> user_type = ContentType.objects.get(app_label='auth',
model='user')
>>> user_type
<ContentType: user>
```

Ejemplo de contentTypes

**Fuente**: docs.djangoproject.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.sessions*

Las sesiones son el mecanismo utilizado por Django para mantener registro del estado entre el sitio y un browser (navegador) particular.

Las sesiones permiten almacenar datos arbitrarios por browser y mantiene estos datos disponibles para el sitio cuando el browser se conecta.

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.sessions*

En la imagen, tenemos un ejemplo donde tomamos valores en una vista, recuperados de la variable request.

Aquí vemos que podemos leer y guardar información en forma de diccionario, con una clave y un valor.

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

Ejemplo de uso de sessions

**Fuente**: docs.djangoproject.com

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.messages*

Es bastante común en las aplicaciones web que necesites mostrar una notificación una sola vez al usuario, después de procesar un formato o algún otro tipo de input.

Para esto Django provee un soporte completo de mensajes basado en cookies y sesiones, tanto para usuarios anónimos como autenticados.

El framework de mensajes te permite almacenar temporalmente mensajes de un uso. Todos los mensajes son clasificados con algún nivel determinado de prioridad (como información, error, advertencia.)

# Aplicaciones Pre-Instaladas de Django

*django.contrib.messages*

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    do_something_with_the_message(message)
```

Guardando y recuperando un mensaje

**Fuente**: docs.djangoproject.com

{desafío} latam_

# Aplicaciones Pre-Instaladas de Django

## *django.contrib.staticfiles*

Es una colección de archivos estáticos de cada una de las aplicaciones que componen el proyecto en una sola ubicación que puede ser configurada fácilmente en producción.

Posee una serie de variables de configuración:

- STATIC_ROOT
- STATIC_URL
- STATICFILES_DIRS
- STATICFILES_STORAGE
- STATICFILES_FINDERS

```
STATICFILES_DIRS = [
    "/home/special.polls.com/polls/static",
    "/home/polls.com/polls/static",
    "/opt/webfiles/common",
]
```

Configuración de staticfiles_dirs

**Fuente**: docs.djangoproject.com

# Ejercicio guiado

# Operando con modelos


{desafío}
latam_

# Ejercicio guiado

## *Operando con modelos*

En este ejercicio crearemos un proyecto y generamos operaciones sobre datos en el modelo user.
Para esto utilizaremos la base de datos por defecto que es sqlite y para realizar los ejercicios utilizaremos la shell de django.

{desafío} latam_

# Ejercicio guiado

*Operando con modelos*

1. Iniciamos un nuevo entorno virtual y creamos un proyecto nuevo de django llamado **proyecto_capitulo_2**

```
python -m venv .\venv
.\venv\Scripts\activate.bat

pip install django
django-admin startproject proyecto_capitulo_2
```

# Ejercicio guiado

## *Operando con modelos*

2. Creamos una aplicación llamada **testadl**

```
cd proyecto_capitulo_2
python manage.py startapp testadl
```

3. Agregamos la nueva aplicación a **INSTALLED_APPS**

```
31  # Application definition
32
33  INSTALLED_APPS = [
34      'django.contrib.admin',
35      'django.contrib.auth',
36      'django.contrib.contenttypes',
37      'django.contrib.sessions',
38      'django.contrib.messages',
39      'django.contrib.staticfiles',
40      'testadl'
41  ]
```

# Ejercicio guiado

## *Operando con modelos*

4. Ahora volvemos a la terminal. En este caso, no hemos creado modelos, por lo tanto no tenemos que ejecutar makemigrations, solo ejecutamos migrate para reflejar los modelos por defecto del Django en la base de datos.

```
python manage.py migrate
```

5. Ahora, desde la terminal nos iremos a la shell de Django.

```
python manage.py shell
```

# Ejercicio guiado
## *Operando con modelos*

6. Vamos a listar los campos disponibles que tenemos en el modelo User, para eso utilizaremos "_meta".
   El modelo _meta del API de django ORM, permite a otros componentes como queries, admin, formularios conocer las capacidades de un determinado modelo. este modelo tiene 2 métodos:

- Objeto._meta.get_field("nombre_del_campo"), el cual obtiene la información relacionada a este campo, como su tipo de dato.
- Objeto._meta.get_fields() el cual retorna una lista con todos los campos disponibles en el modelo.

```
>>> from django.contrib.auth.models import User
>>>
>>> campos = User._meta.get_fields()
>>> for c in campos:
...     print(c.name)
```

# Ejercicio guiado
## *Operando con modelos*

7. Presionamos dos veces enter y obtenemos:

```
logentry
id
password
last_login
is_superuser
username
first_name
last_name
email
is_staff
is_active
date_joined
groups
user_permissions
```

# Ejercicio guiado
## *Operando con modelos*

8. Entonces crearemos 2 usuarios, usando el siguiente código:

```
>>> u1 = User(username='jdoe', first_name='John', last_name='Doe', email='jdoe@mail.com')
>>> u1.save()
>>> u2 = User(username='ltorvalds', first_name='Linus', last_name='Torvalds',email='ltorvalds@mail.com')
>>> u2.save()
```

9. Listamos los usuarios existentes:

```
>>> users = User.objects.all()
>>> for user in users:
...     print(user)
...
jdoe
ltorvalds
```

# Ejercicio guiado

## *Operando con modelos*

11. Eliminamos al usuario Juan Doe y listamos los usuarios existentes, solo debería quedar uno.

```
>>> User.objects.filter(username='jdoe').delete()
(1, {'auth.User': 1})
>>> users = User.objects.all()
>>> for user in users:
...     print(user)
...
ltorvalds
```

# ¿Comó nos ayuda a manejar los datos en Django?

# ¿Que desventajas tenemos usando el ORM de Django?

{desafío} latam_

# ¿En qué nos beneficia las aplicaciones pre-instaladas en Django?

# ¿Que beneficios trae tratar el modelo como un objeto?

# Próxima sesión...

- *Guía de ejercicios*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales