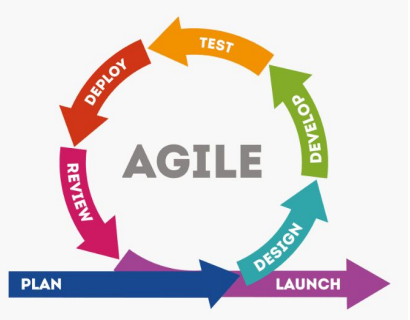
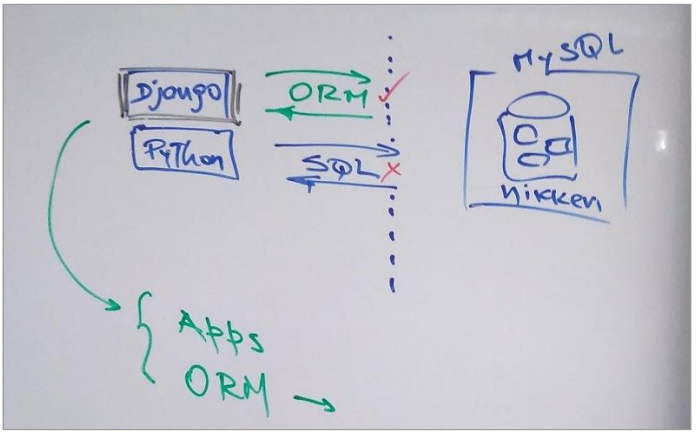
BDII Lab

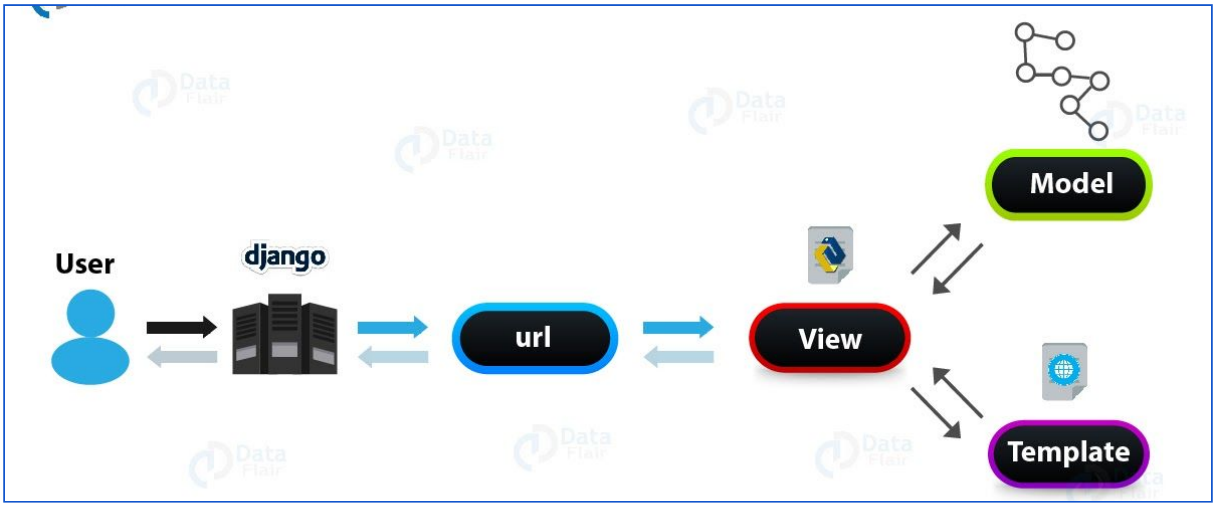
Aplicación Python/Django que calcula la ganancia mensual de los Asesores.

1. Modelado UML: Usar como referencia el diagrama de clases (abajo), el cual debe ser parte de un diagrama de paquetes. Utilizar StarUML
2. Mapeo a Django (modelo de datos)
3. Creación del entorno Django y el proyecto (usar Visual Studio Code - VSC)
4. Creación de aplicación Django
5. Usar el Shell Django para poblar la BD, por medio de una transacción que inserte las Abono (Cabecera y detalle). El motor de BD puede ser SQLite o MySQL

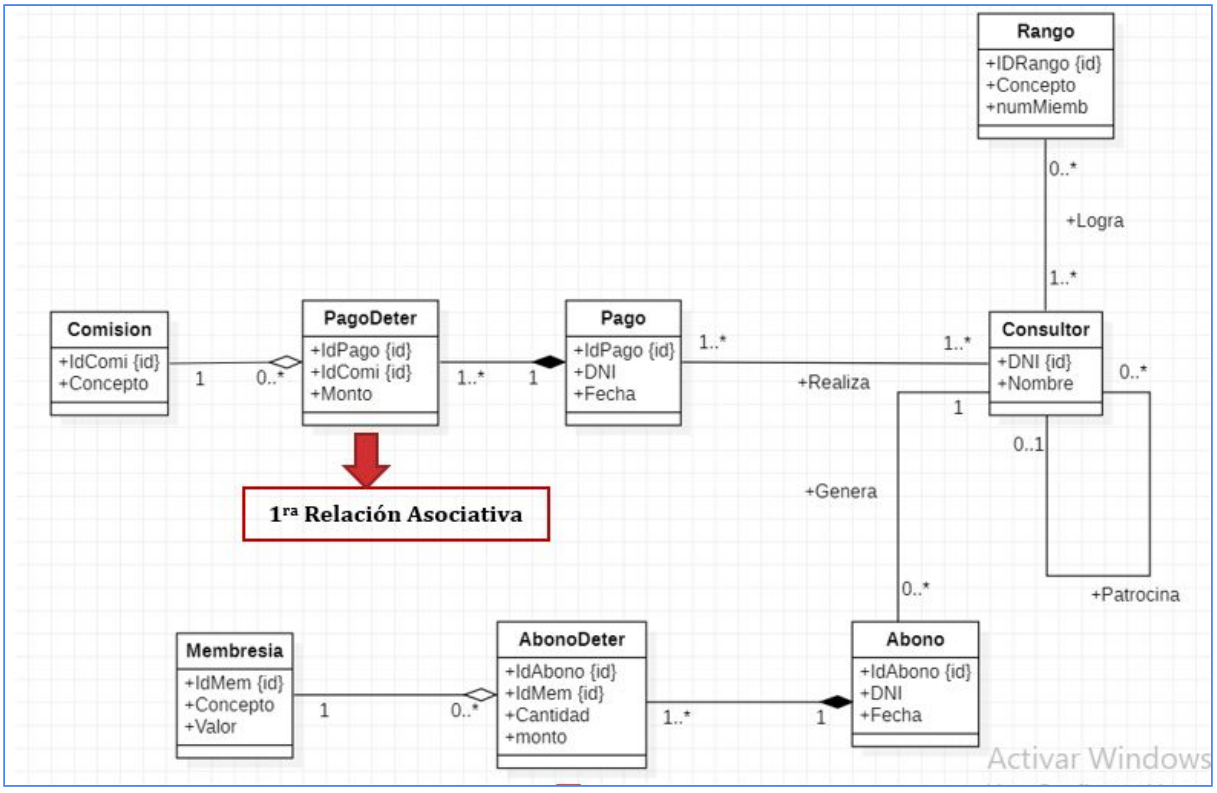


6. Elaborar una función para calcular las comisiones, tomando los datos de la BD y guardar el resultado en esta.





https://data-flair.training/blogs/django-architecture/



APLICACIONES WEB CON DJANGO/PYTHON
https://www.hektorprofe.net/curso/curso-django-principiantes

PRIMEROS PASOS

Django

- Django es un framework web creado en 2003, programado en Python, gratuito y libre.
- Fue liberado en 2005 y desde 2008 cuenta con su propia fundación para hacerse cargo de su desarrollo.
- Es utilizado por multitud de grandes empresas como Instagram, Disqus, Pinterest, Bitbucket, Udemy y la NASA
- Promueve el desarrollo ágil y extensible basado en un sistema de componentes reutilizables llamados "apps".
- Algunas de sus mejores características son su mapeador ORM para manejar la base de datos como conjuntos de objetos junto a un panel de administrador autogenerado, así como un potente sistema de plantillas extensibles.
- Funciona con distintos gestores de base de datos SQL y es ideal para manejar autenticación de usuarios, sesiones, operaciones con archivos, mensajería y plataformas de pago.
- No es la mejor alternativa para manejar microservicios, ni tampoco servicios que requieran ejecutar múltiples procesos, como las aplicaciones de big data y plataformas con sockets en tiempo real.

Creación del entorno virtual con pipenv

- Crear una carpeta **nikken** para el proyecto
- Abrirlo en VSC para que se inicie la configuración
- Abrir el terminal e instalar **pipenv**:
  pip install pipenv

Crear el entorno virtual, instalando Django directamente en la carpeta del proyecto:

```
pipenv install django
```

Creación del Proyecto

```
pipenv run django-admin startproject nikken
```

Probar a poner el servidor web en marcha:

```
cd nikken
pipenv run python manage.py runserver
```

Ahora se puede acceder a la URL y ver el proyecto:
http://127.0.0.1:8000/

Crear un script en el Pipfile para simplificar la puesta en marcha:

Pipfile

```
[scripts]
server = "python manage.py runserver"
```

Ahora se puede ejecutar el servidor con:

```
pipenv run server
```

Configuración básica del proyecto

Configuración del proyecto que se realiza en los archivos de la carpeta con el mismo nombre que el proyecto .....

CREANDO LA APLICACIÓN

Apps

Cada app de Django es independiente, contiene sus propias definiciones, templates y ficheros estáticos. Necesitamos crear como mínimo una app para empezar a trabajar y, cómo vamos a estar desarrollando una pequeña aplicación de prueba, creamos esta app con el nombre, **ninja**

```
pipenv run python manage.py startapp ninja
```

Se crea una carpeta con el nombre de la app en nuestro proyecto y dentro, muchos archivos que, iremos revisando poco a poco. Por ahora, sólo falta activar la app en el settings.py para poder utilizarla, para ello la añadimos en la lista:

nikken\settings.py

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

Modelos

Los **modelos** de Django son **clases** donde se define la estructura de los datos para almacenarlos en la base de datos (nikken). Entre las clases iniciales se tiene: Asesor, Producto, OC y OCDetalle

ninja/models.py

```
from django.db import models
class Asesor(models.Model):
    pass
```

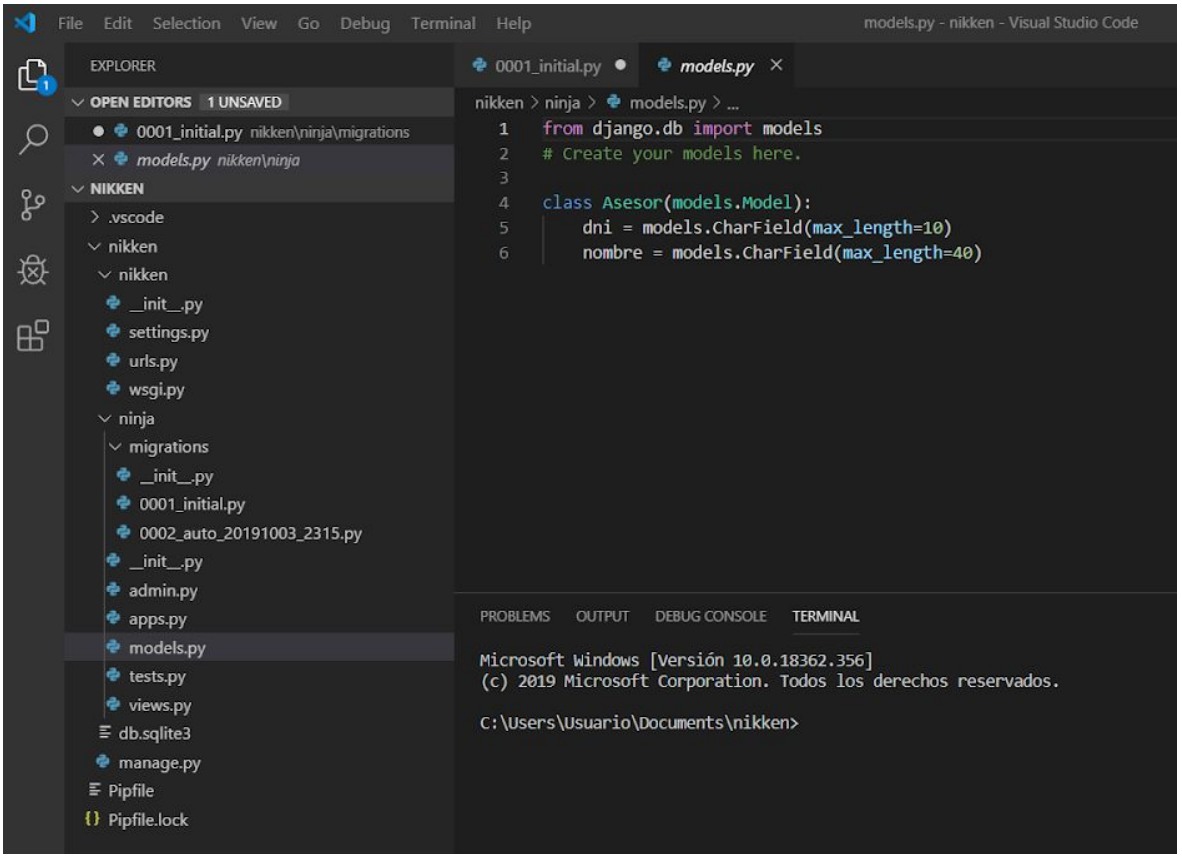
Una vez que se tiene el modelo, equivalente a la estructura de una tabla en la base de datos, se tiene que crear una **migración** con un registro de los cambios:

```
pipenv run python manage.py makemigrations
```

Una vez hecha la migración, se aplica los cambios en la base de datos:

```
pipenv run python manage.py migrate
```

Al ser la primera vez que migramos y al tener activadas las apps genéricas del admin, de autenticación y de sesiones, se crean varias tablas en la BD. La misma que se encuentra en la raíz del proyecto con el nombre **db.sqlite3**. Este archivo (binario) se puede consultar y editar con un programa como DB Browser for SQLite. Dentro se encuentran todas las tablas de la BD. Ahora, tenemos el modelo Asesor listo para añadir filas.



Administrador

Existen tres formas de manejar las tablas de la base de datos:

1. ~~A través del código al responder una petición.~~

2. A través del panel de administrador autogenerado.
3. A través de la shell de Django, un intérprete de comandos.

En este curso NO veremos la primera forma, se hará uso del panel de administrador y experimentar un poco con el shell. Para acceder al panel sólo hay que acceder a la URL /admin. Esta dirección no es casual, está definida así dentro del fichero urls.py del proyecto.

http://127.0.0.1:8000/admin/login/?next=/admin/

Nos pedirá identificarnos: Usuario y contraseña. Este usuario no puede ser cualquiera, debe ser un usuario con permisos para acceder al administrador y, como no tenemos, vamos a tener que crearlo. Para crear nuestro primer superusuario lo haremos desde el *terminal*:

```
pipenv run python manage.py createsuperuser
```

Una vez creado y con el servidor en marcha, se accede al panel, donde se encuentra la sección: Autenticación y autorización. Esta sección corresponde a la app **auth** de Django y contiene dos modelos, uno para manejar grupos de permisos y otro con los propios usuarios.

Como ves, sin hacer nada ya contamos con un panel donde podemos manejar usuarios cómodamente y añadirles permisos para manejar otras apps. Sin embargo, nuestra app ninja todavía no aparece, para ello hay que activar los modelos que queremos manejar en el administrador. Así que, hay que configurar el admin para el modelo Asesor.

ninja/admin.py

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

Con este cambio aparece nuestra app y podemos empezar a crear, editar y borrar asesores. Con esto ya lo tenemos, pero antes veamos algunas opciones para personalizar los campos que se muestran en el archivo de modelos: ninja/models.py

```
class Asesor(models.Model):
    dni = models.CharField(max_length=10, verbose_name="D.N.I.")
    nombre = models.CharField(max_length=40, verbose_name="Nombres")
```

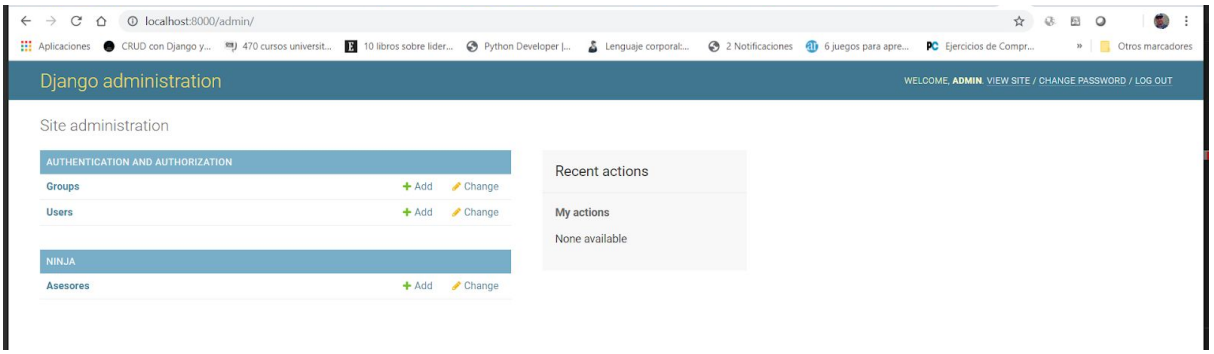
Con eso podemos mostrar un nombre diferente en los campos y si queremos mostrar un nombre de modelo diferente lo haremos así:

```
class Meta:
  verbose_name = "Asesor"
  verbose_name_plural = "Asesores"
```

Por último, para listar las entradas mostrando su nombre y no esa referencia rara al objeto que aparece, podemos hacerlo sobreescribiendo el método string del modelo:

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

Obviamente hay mil cosas más que se pueden configurar, pero hablaremos de eso en futuros cursos. Esto es más que suficiente por ahora, crea un par más de filas y seguimos.



Shell (interfaz) de Django

El shell de Django es un intérprete de comandos que, nos permite ejecutar código directamente en el backend, algo muy útil para hacer pruebas y consultas.

```
pipenv run python manage.py shell
```

Una vez dentro, podremos escribir las instrucciones que queramos siempre que sigamos una lógica. Por ejemplo, si queremos consultar los *asesores* de la base de datos, antes tendremos que importar el modelo Asesor y luego utilizar la sintaxis para hacer la consulta:

```
from ninja.models import Asesor
Asesor.objects.all()
```

Esta instrucción objects.all() permite recuperar todas las filas que hay en la tabla Asesor y las almacena en una lista especial de Django llamada QuerySet. También podemos recuperar el primer y último registro muy fácilmente:

```
Asesor.objects.first()
Asesor.objects.last()
```

O una fila a partir de su identificador, un número automático que maneja Django internamente:

```
Asesor.objects.get(id=1)
```

Todo lo que estamos haciendo son consultas a la base de datos, pero se encuentran abstraídas gracias a la API de acceso, al mapeador **ORM** que proporciona Django, donde

cada fila se puede manejar como un objeto. De hecho vamos a crear una fila y a manipularla un poco:

```
asesor = Asesor.objects.create(dni="10372655", nombre="Melchorita Luz")
```

Al ejecutar lo anterior tendremos nuestra fila creada en la base de datos, podemos consultar el administrador, pero también la tendremos guardada en la variable asesor. Podemos editar sus atributos como como cualquier objeto:

```
asesor.nombre = "Mel Luz"
```

Eso sí, tendremos que llamar al método save() del objeto para guardar los cambios:

```
asesor.save()
```

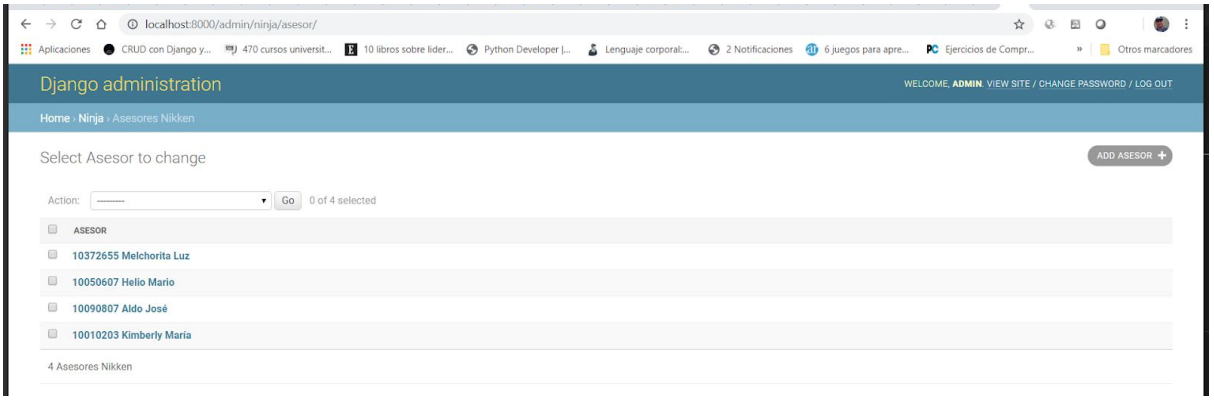
Por último, si quisiéramos borrar esta fila que tenemos en la variable, podemos hacerlo con el método delete():

```
asesor.delete()
```

Y para salir del shell simplemente llamaremos la función quit() desde el terminal:

```
quit()
```

Como ven, se está interactuando con la base de datos a través de objetos y métodos, una forma muy sencilla y cómoda de manejar nuestras tablas.



Vistas y Templates/URLs

Todo lo que hemos hecho hasta ahora no ha implicado todavía el manejo de peticiones (request) y respuestas (response). Las peticiones son las diferentes URL que los clientes piden ver, por ejemplo **/** es la portada y /ninja/ podría ser nuestra aplicación.

Para manejar estas peticiones, Django utiliza el siguiente flujo:

1. El cliente hace la petición a una dirección definida en el urls.py.
2. Esa dirección está enlazada a una **vista**, una función definida en el archivo views.py y que contiene la lógica que procesa la petición, como por ejemplo, consultar el modelo Asesor, para ver qué filas hay en la base de datos.
3. Por último, estos datos se renderizan sobre un **template** HTML y se envían al cliente para que éste vea el resultado de la petición.

Este flujo de trabajo se conoce en Django como **Patrón MVT (Modelo - Vista - Template)** y permite separar muy bien cada proceso de los demás. Para ver todo esto en acción vamos a programar la vista de nuestra aplicación y que devuelva un simple texto plano:

ninja/views.py

```
from django.shortcuts import render, HttpResponse
def home(request):
    return HttpResponse("Bienvenido al Portal de Bienestar y Salud")
```

Ahora tenemos que enlazar esta vista a una URL para poder hacer la petición:

nikken/urls.py

```
from django.contrib import admin
from django.urls import path
from ninja.views import home

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

Tan simple como esto y si accedemos a la raíz de nuestro sitio ya deberíamos ver cómo se devuelve el texto plano que devolvemos. Sin embargo, la gracia es **renderizar** un template HTML bien estructurado, así que vamos a crear uno para nuestra portada.

Presta mucha atención, para crear un template dentro de la app ninja tenemos que crear una carpeta templates en la app y dentro otra carpeta con el mismo nombre que la app, en nuestro caso ninja.

Se hace de esta forma porque Django carga en memoria todas las carpetas templates de las apps, unificándolos en el mismo sitio. En cualquier caso dentro ya podremos crear nuestra plantilla HTML para renderizarla:

templates/ninja/home.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>NIKKEN</title>
</head>
<body>
    <h1>Bienvenidos al Portal de Bienestar y Salud</h1>
</body>
</html>
```

Por último, vamos a cambiar la vista para en lugar de devolver el texto plano *renderice* esta plantilla HTML que hemos creado:

ninja/views.py

```
from django.shortcuts import render, HttpResponse

def home(request):
    return render(request, "ninja/home.html")
```

Listo, ya hemos completado la parte de la vista y el template. En la siguiente lección incorporaremos una consulta a la base de datos, a través del modelo Asesor y devolveremos las filas al template para renderizarlas.

Variables de contexto

Para recuperar las filas se tiene que cargar el modelo y hacer exactamente lo que se hizo cuando se experimentó con el shell:

ninja/views.py

```
from django.shortcuts import render, HttpResponse
from .models import Asesor

def home(request):
    asesor = Asesor.objects.all()
    return render(request, "ninja/home.html")
```

Una vez tenemos las entradas recuperadas tendremos que enviarlas al template y eso lo haremos usando un diccionario de contexto:

```
return render(request, "ninja/home.html", {'asesores': asesor})
```

Los datos que enviamos en el diccionario de contexto se pueden recuperar en el template usando template tags, una de las funcionalidades más atractivas de Django:

templates/ninja/home.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>NIKKEN</title>
</head>
<body>
    <h1>Bienvenidos al Portal de Bienestar y Salud</h1>
    <p>Estos son los Asesores.</p>
    {{ asesores }}
</body>
</html>
```

Lo más increíble de todo es que podemos usar unos template tags especiales que permiten ejecutar lógica de programación en los templates, por ejemplo, para recorrer todas las entradas almacenadas en la QuerySet asesores:

```
{% for asesor in asesores %}
<div>
   <h2>{{ asesor }}</h2>
</div>
{% endfor %}
```

Por defecto se nos muestra el DNI - Nombre porque es lo que devolvemos al sobreescribir el método str del modelo, pero lo que tenemos son objetos, por lo que podríamos acceder a sus campos específicos:

```
{% for asesor in asesores %}
<div>
   <h2>{{ asesor.dni }}</h2>
   <p>{{ asesor.nombre }}</p>
</div>
{% endfor %}
```

Con esto hemos completado el flujo del patrón Modelo - Vista - Template, recuperamos los datos del modelo y los enviamos al template a través de la vista. Django se basa siempre en esa idea.

Páginas dinámicas

En esta última lección vamos a introducir el concepto de las páginas dinámicas, añadiendo una nueva página para visualizar las filas individualmente, en lugar de mostrar todo su contenido en la lista.

La clave está en pasar un argumento en la URL, a través del cual podamos recuperar la fila para renderizarla. ¿Cuál es ese argumento? Usualmente el identificador de la fila (su id), al que también se puede acceder con el nombre pk - primary key:

ninja/views.py

```
def asesor(request):
   ases = Asesor.objects.get(id=?)
   return render(request, "ninja/asesor.html", {'asesor': ases})
```

Se ha creado la vista y se cargará el template asesor.html enviándole el objeto asesor en el diccionario de contexto. Pero falta lo más importante, una forma de recuperar el identificador de la fila.

Esto se maneja en dos partes, primero en la URL definiendo un parámetro:

nikken/urls.py

```
from django.contrib import admin
from django.urls import path
from blog.views import home, asesor

urlpatterns = [
    path('', home),
    path('ninja/<id>', asesor),
    path('admin/', admin.site.urls),
]
```

Y luego en la vista, creando ese parámetro como si formara parte de la función:

ninja/views.py

```
def asesor(request, id):
    ases = Asesor.objects.get(id=id)
    return render(request, "ninja/asesor.html", {'asesor': ases})
```

Con esto ya lo tenemos, aunque si accedemos nos dará error de “template no encontrado”, porque aún no lo hemos creado:

templates/ninja/asesor.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{asesor.dni}}</title>
</head>
<body>
    <h1>{{asesor.dni}}</h1>
    <p>{{asesor.nombre}}</p>
        <a href="/">Volver al Inicio</a>
</body>
</html>
```

Ahora sólo hay que añadir un enlace a los nombres para moverse a la fila (Asesor):

ninja/templates/ninja/home.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>NIKKEN</title>
</head>
```

```
<body>
   <h1>Bienvenidos al Portal de Bienestar y Salud</h1>
   <p>Estos son los asesores.</p>
   {% for asesor in asesores %}
   <div>
      <p> <a href="/ninja/{{asesor.id}}">{{ asesor.nombre }}</a></p>
   </div>
   {% endfor %}
</body>
</html>
```

Nuestra web ahora tendrá tantas páginas como filas tengamos en Asesores, pero nosotros solo hemos creado la plantilla base. O lo que es lo mismo, tenemos una web generada dinámicamente, a partir de los datos que hay en la BD.