TI2041

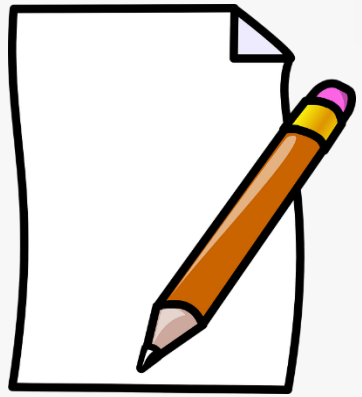
PROGRAMACIÓN BACK END

Unidad 1: Tecnología del Lado del Servidor

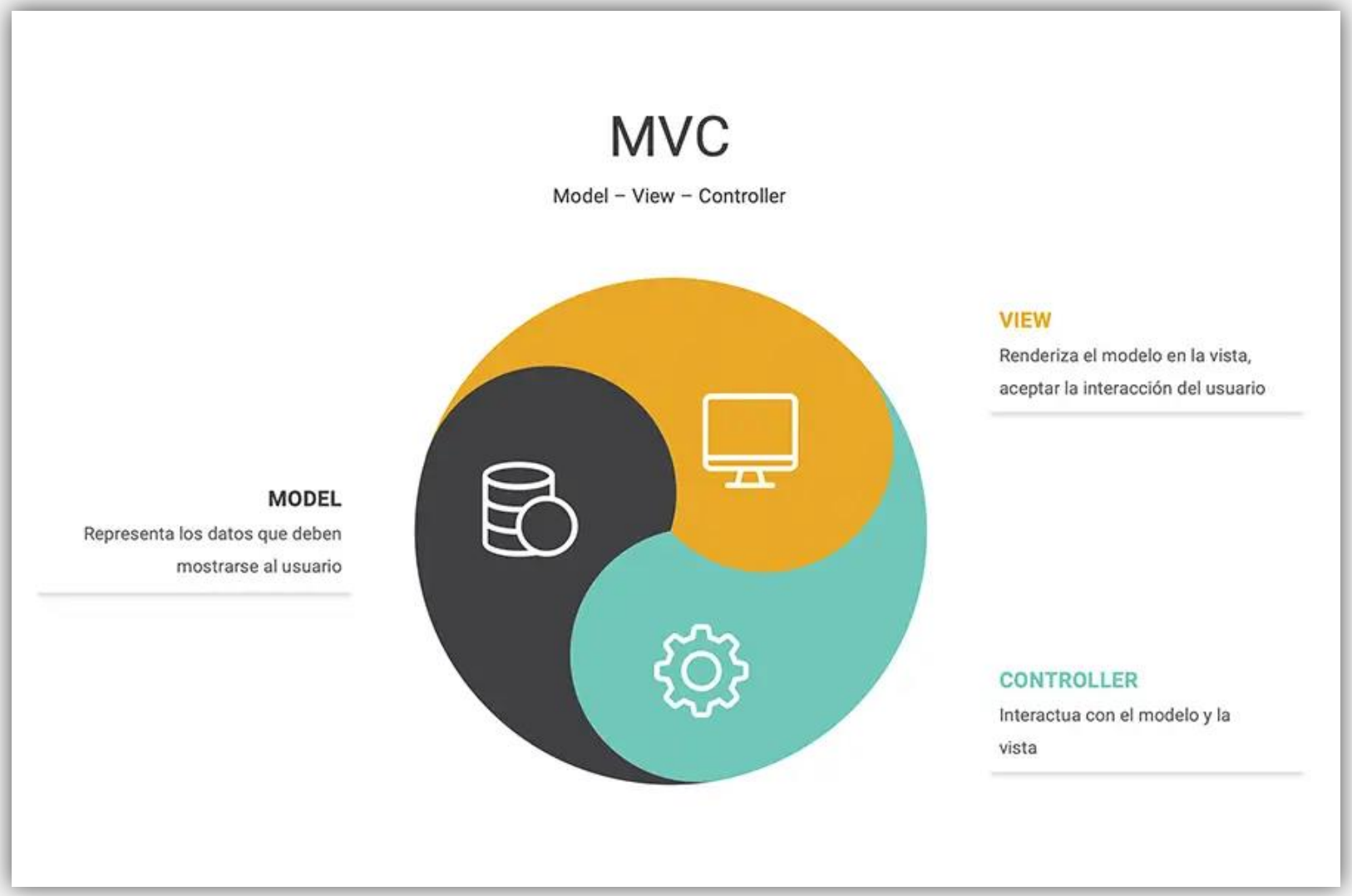
Clase 3

APRENDIZAJE ESPERADO

Realiza un programa del lado del servidor, de acuerdo a la sintaxis del lenguaje.



MOTIVACIÓN

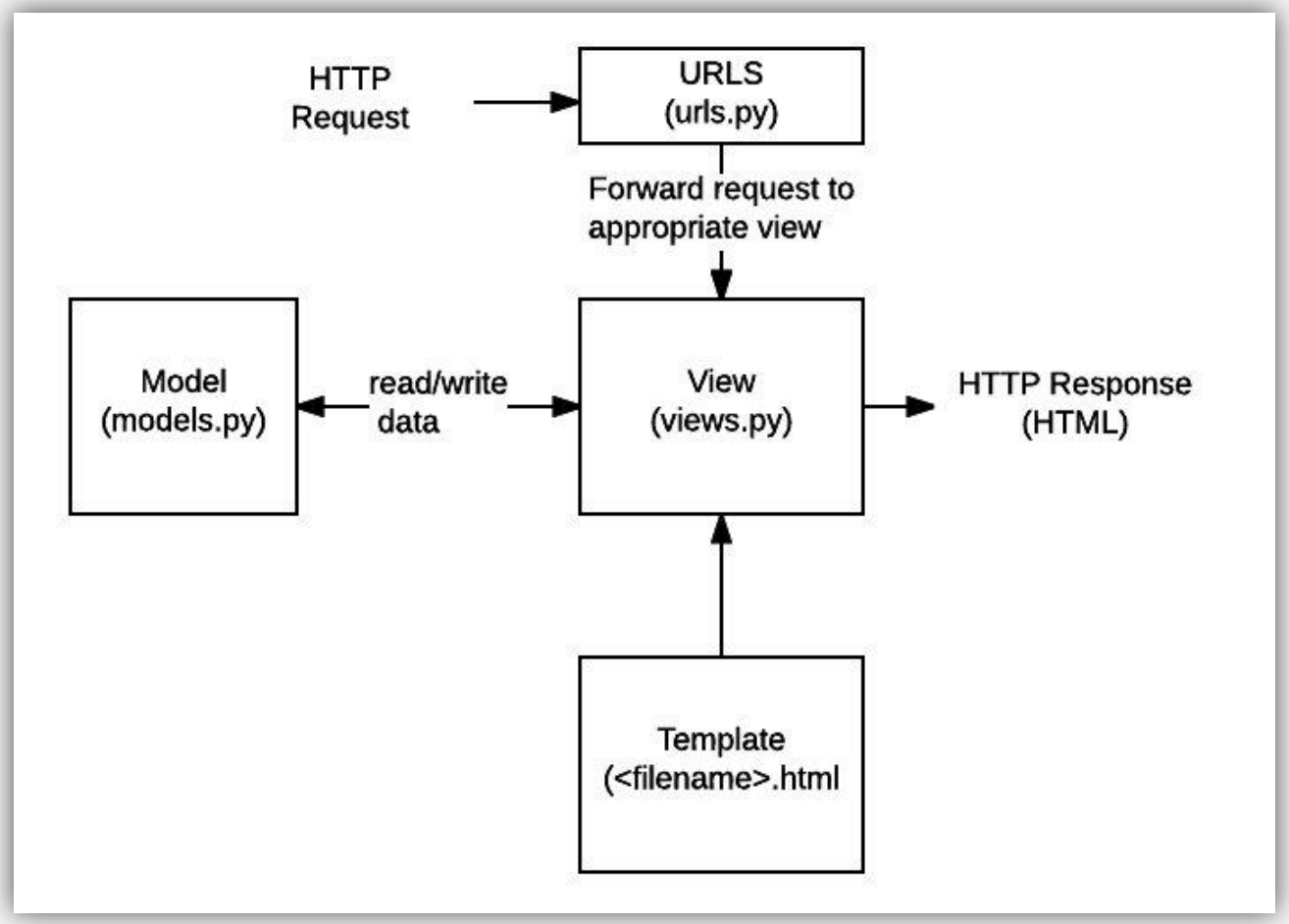

MVC
Model – View – Controller
MODEL
Representa los datos que deben mostrarse al usuario
VIEW
Renderiza el modelo en la vista, aceptar la interacción del usuario
CONTROLLER
Interactua con el modelo y la vista

EN ESTA CLASE

- MVC en Django
  - URLs
  - Vistas
  - Modelos
  - Templates
- Formularios
  - El objeto Request
- Ejemplo

MVC EN DJANGO

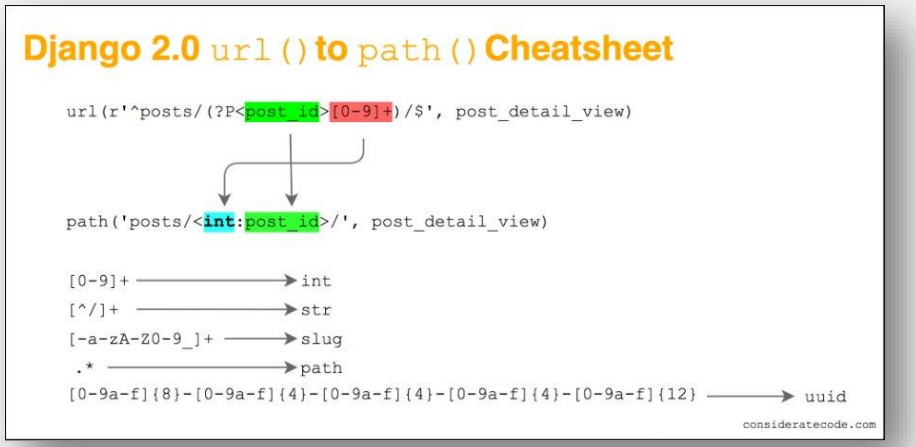
- Django utiliza una variante del Model-View-Controller, que se llama **Model-View-Template**.
- MVT utiliza **4 elementos**:
  - URLs
  - Vistas
  - Modelos
  - Templates



Más información: https://docs.hektorprofe.net/django/web-personal/patron-mvt-modelo-vista-template/

MVC EN DJANGO

- **URLs**:
  - Es el conjunto de mapeos de direcciones únicas a las funciones de visualización (vistas) para cada recurso.
  - Este mapeador permite redirigir los requerimientos HTTP a la vista apropiada basándose en la URL de petición.
  - El Mapeador URL puede también emparejar patrones de cadenas o dígitos específicos, los cuales pasan a la función de visualización como datos.



MVC EN DJANGO

- **URLs** (cont):
  - Para configurar el mapeador de URLs, cada app en Django tiene su propio archivo **`urls.py`** que contiene lo siguiente:

```python
from django.urls import path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<int:year>/", views.year_archive),
    path("articles/<int:year>/<int:month>/", views.month_archive),
    path("articles/<int:year>/<int:month>/<slug:slug>/", views.article_detail),
]
```

  - En este caso, se utiliza la función **`path()`** para indicar el mapeo de cada una de las vistas (segundo parámetro).

MVC EN DJANGO

- **URLs** (cont):
  - También es posible utilizar expresiones regulares en la dirección mapeada con la función **`re_path()`**:

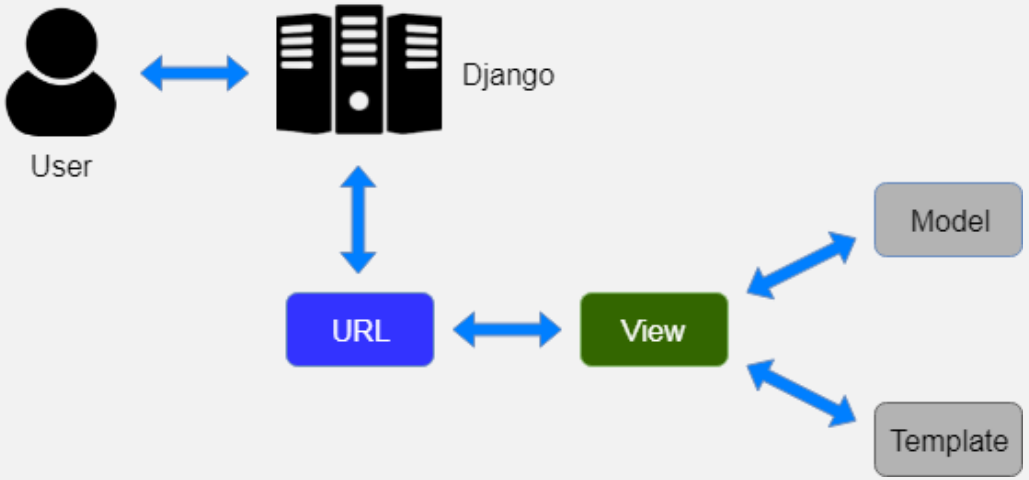
```python
from django.urls import re_path

urlpatterns = [
    re_path(r"^blog/(page-([0-9]+)/)?$", blog_articles),  # bad
    re_path(r"^comments/(?:page-(?P<page_number>[0-9]+)/)?$", comments),  # good
]
```

  - En este caso, la expresión permite mayor control sobre las URLs permitidas para cada redireccionamiento.

Visita https://regex101.com/ para mayor información sobre Expresiones Regulares

MVC EN DJANGO

- **Views**:
  - Una vista es una función de gestión de peticiones que recibe requerimientos HTTP y devuelve respuestas HTTP.
  - Las vistas acceden a los datos (a través de los modelos).
  - Delegan el formateo de las respuestas a las plantillas (templates).



MVC EN DJANGO

- **Views** (cont):
  - Una vez que está mapeada la vista, se puede construir algo muy simple como ésto:

```python
from django.http import HttpResponse
import datetime


def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

  - La función que maneja el requerimiento dentro del archivo views.py es la vista asignada. En este caso, es **`current_datetime()`** la que presentará simplemente un mensaje con la fecha y hora actual.

MVC EN DJANGO

- **Views** (cont):
  - Otra gracia que tienen las vistas es que pueden manejar errores HTTP en sus respuestas:

```python
from django.http import HttpResponse, HttpResponseNotFound


def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound("<h1>Page not found</h1>")
    else:
        return HttpResponse("<h1>Page was found</h1>")
```

  - De esta forma, no solo se puede indicar al browser que hay un error (que interpreta el usuario) sino que el mismo navegador es el que lo entiende.

MVC EN DJANGO

- **Views** (cont):
  - Para enviar un Error 404 tradicional y hacer que se encargue el servidor web, basta que se utilice la excepción **`Http404`**:

```python
from django.http import Http404
from django.shortcuts import render
from polls.models import Poll


def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, "polls/detail.html", {"poll": p})
```

MVC EN DJANGO

- **Views** (cont):
  - Otra función interesante que tienen las vistas, son el uso de decoradores (*decorators*).
  - Los *decorators* son sentencias que modifican la ejecución de la función de procesamiento a ciertas características:

```python
from django.views.decorators.http import require_http_methods


@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

  - El *decorator* **`require_http_methods()`** permite filtrar el acceso solo a esos dos métodos HTTP.

MVC EN DJANGO

- **Views** (cont):
  - Ya vimos anteriormente que con HttpResponse podemos enviar HTML como un string directo desde el código de la vista.
  - Otra función relevante (y no menor) es la función render(), la cual permite procesar una plantilla y enviarla como respuesta:

```python
from django.shortcuts import render


def my_view(request):
    # View code here...
    return render(
        request,
        "myapp/index.html",
        {
            "foo": "bar",
        },
        content_type="application/xhtml+xml",
    )
```

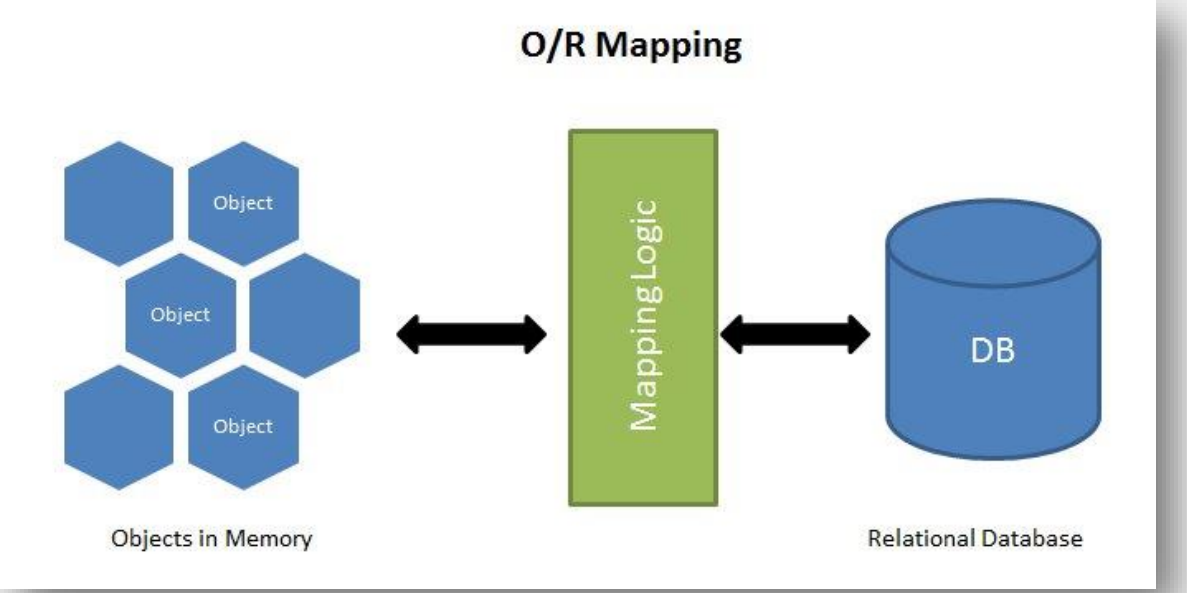
MVC EN DJANGO

- **Views** (cont):
  - Otra función que se utiliza es el **`redirect()`**, la cual permite redirigir el control de la app a otra vista, dirección o recurso HTTP

```python
from django.shortcuts import redirect


def my_view(request):
    ...
    return redirect("some-view-name", foo="bar")
def my_view(request):
    ...
    return redirect("/some/url/")
def my_view(request):
    ...
    return redirect("https://example.com/")
```

MVC EN DJANGO

- **Models**:
  - Los modelos son objetos de Python que definen una estructura de datos.
  - Proporcionan mecanismos para gestionar y consultar los datos (CRUD).
  - Funcionan como un ORM ya que abstraen la base de datos del resto de la aplicación.



MVC EN DJANGO

- **Models** (cont):
  - Hasta ahora, los archivos Django contienen funciones y estructuras de listas.
  - Sin embargo, los modelos se organizan en clases:

```python
from django.db import models


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

  - Y es equivalente a decir en la base de datos:

```sql
CREATE TABLE myapp_person (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

MVC EN DJANGO

- **Models** (cont):
  - Crear las clases para los modelos no es lo único, ya que, como Django mantiene la base de datos en el ORM, ésta se debe "sincronizar" con los modelos.
  - Se debe de editar el archivo **`settings.py`** que se encuentra en la carpeta del sitio creado y agregar la aplicación a la lista **`INSTALLED_APPS`**:

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

  - Con esto realizado, se puede ejecutar el proyecto con la opción makemigrations, la cual crea, en SQLite, la base de datos.

MVC EN DJANGO

- **Models** (cont):
  - Utilizar los modelos es muy sencillos, ya que abstraen el proceso de manipulación de datos de la base de datos.
  - Por ejemplo:

```python
from django.db import models


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

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

MVC EN DJANGO

- **Models** (cont):
  - También podemos relacionar modelos para que dependan entre sí:

```python
from django.db import models


class Manufacturer(models.Model):
    # ...
    pass


class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

  - En este caso no necesitamos depender ni conocer el valor de la llave primaria, ya que queda relacionado a nivel de objetos.
  - Luego, el ORM se entenderá con la base de datos para la relación "física" entre ellos.

MVC EN DJANGO

- **Models** (cont):
  - Los métodos para utilizar los modelos son:

Método	Descripción
`create(...)`	Crea un objeto en particular en una colección.
`set(...)`	Actualiza algunos o todos los atributos de un objeto en particular.
`add(...)`	Agrega un objeto existente a una colección.
`all()`	Obtiene los objetos de una colección.
`remove(...)`	Elimina uno o más objetos de una colección.
`clear()`	Elimina todos los objetos de una colección.
`filter(...)`	Filtra los objetos de una colección.
`get(...)`	Obtiene los objetos que cumplen con cierto criterio.

Más información en https://docs.djangoproject.com/es/4.2/topics/db/models/

MVC EN DJANGO

- **Templates**:
  - Las plantillas son ficheros de texto que definen una estructura (como una página HTML), con marcadores de posición que se utilizan para representar el contenido real.
  - Un vista puede crear dinámicamente una página usando una plantilla, rellenándola con datos de un modelo.

```
<html>
<head>
  <title>Available Resturants</title>
  {% csrf_token %}
</head>
<body>
  <h1>WE are here to serve you</h1><br>
  <table>
    {% for choice in a %}
    <tr>
      <td>1.</td>
      <td>{{choice.name}}</td>
    </tr>
    {% endfor %}
  </table>
</body>
</html>
```

MVC EN DJANGO

- **Templates** (cont):
  - Django provee el uso de diferentes motores de plantillas.
  - A pesar del motor, el lenguaje que utilizan las plantillas es HTML, XML, CSV o cualquier tipo de lenguaje basado en texto.
  - Lo que Django requiere dentro de la plantilla es el marcado de los elementos de datos que debe procesar.

MVC EN DJANGO

- **Templates** (cont):
  - Por ejemplo:

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

MVC EN DJANGO

- **Templates** (cont):
  - Uno de los elementos principales es el uso de variables en las plantillas.
  - Las variables pueden ser:
    - Información de una variable
    - Atributo o método de un objeto
    - Índice numérico

```
Hello, {{ name }}
```

MVC EN DJANGO

- **Templates** (cont):
  - También, se pueden usar filtros de acuerdo a otras funciones específicas:
    - Valores por defecto.

```
{{ value|default:"nothing" }}
```

    - Largo de la variable.

```
{{ value|length }}
```

    - Tamaño de la variable.

```
{{ value|filesizeformat }}
```

Más filtros en https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#ref-templates-builtins-filters

MVC EN DJANGO

- **Templates** (cont):
  - Otros elementos importantes son las etiquetas o *tags* que Django incorpora.
  - En estas etiquetas se pueden incorporar instrucciones de programa (Python) para dar lógica a bloques de plantillas.
  - Al igual que los tags HTML, deben tener un inicio y un fin:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

  - La instrucción **`for`** en las plantillas, debe terminar con **`enfor`**.

MVC EN DJANGO

- **Templates** (cont):
  - En este caso, la instrucción **`if`** puede ir acompañada por un **`elif`**, **`else`** y **`endif`** dependiendo del tipo de bloque que queramos construir.

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

  - Y también se pueden poner comentarios.

```
{# greeting #}hello
```

MVC EN DJANGO

- **Templates** (cont):
  - Otro elemento importante es que las plantillas se pueden heredar.
  - Esto quiere decir, que se puede dejar un código de plantilla fijo y luego hacer bloques variables dependiendo de la plantilla específica a partir de ésta.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

MVC EN DJANGO

- **Templates** (cont):
  - Al igual que otros elementos, las configuraciones de las plantillas están en **`settings.py`**.

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            # ... some options here ...
        },
    },
]
```

  - Aquí se configuran los directorios (lista) que correspondan y los motores de plantillas utilizadas.

MVC EN DJANGO

- **Templates** (cont):
  - Luego, para hacer el procesamiento, se puede utilizar la función **`render()`**:

```python
from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    context = {"latest_question_list": latest_question_list}
    return render(request, "polls/index.html", context)
```

  - Esta función no es más que la composición de la función **`render_to_string()`** y **`HttpResponse()`**.

MANIPULACIÓN DE FORMULARIOS

- Manejar formularios es bastante sencillo utilizando el **objeto `request`**.
- Este objeto nos permite acceder a todos elementos que un **requerimiento HTTP** envía hacia el *backend*.
- Particularmente, si tenemos un formulario HTML, podremos acceder a través de este elemento a **todos los componentes del formulario**.
  - Si fue enviado por método GET >> **`request.GET`**
  - Si fue enviado por método POST >> **`request.POST`**

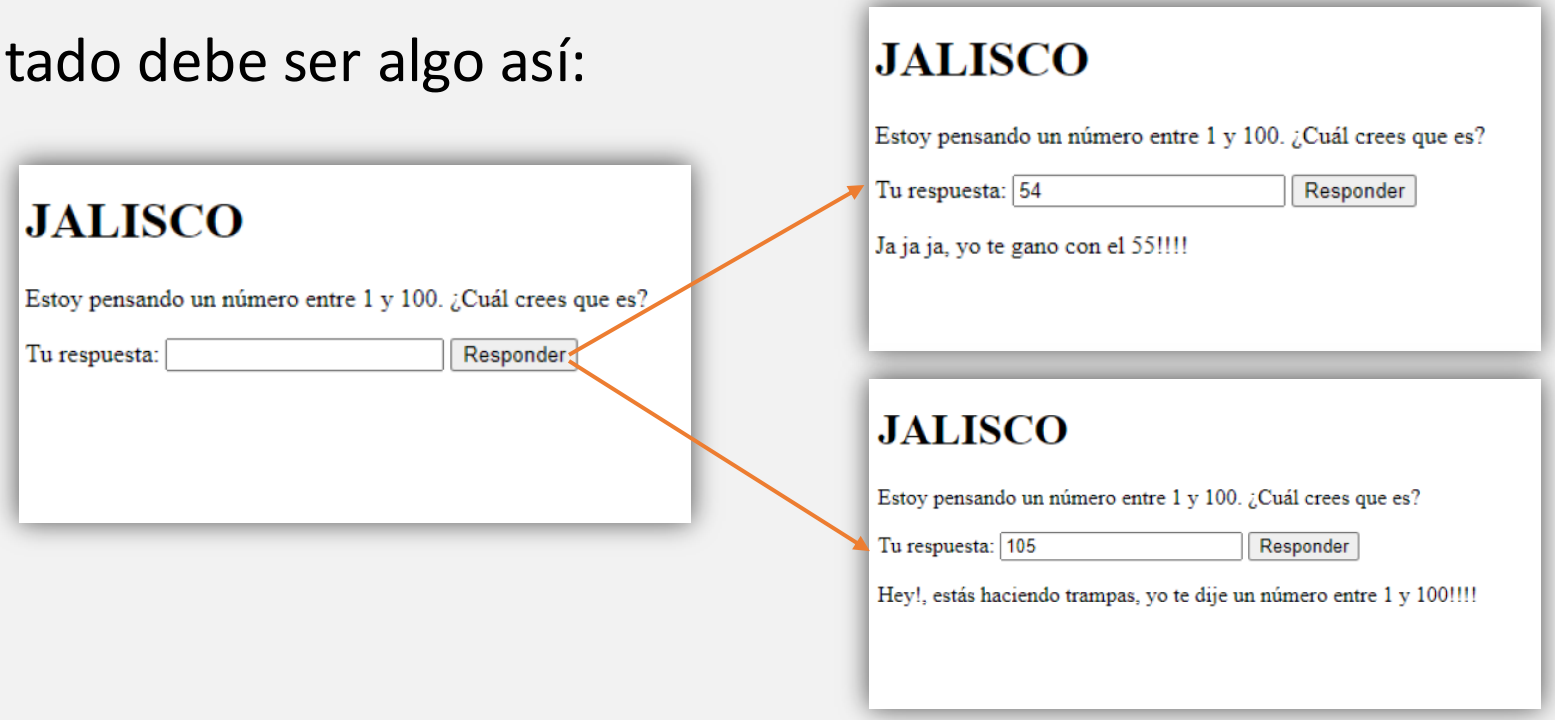
MANIPULACIÓN DE FORMULARIOS

- Ejemplos de uso:

Instrucción	Descripción	Ejemplo
`method`	Obtiene el tipo de petición, si es GET o POST.	`if request.method == 'GET':`
`GET.get(...)`	Obtiene el valor de una variable por GET.	`n = request.GET.get('n')`
`POST.get(...)`	Obtiene el valor de una variable por POST.	`n = request.POST.get('n')`

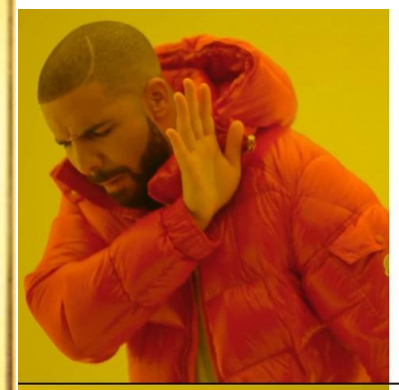
EJEMPLO

- Realizaremos el clásico **juego Jalisco** (algo que me gusta mostrar por su interacción con el usuario).
  - Lo construiremos por completo utilizando vistas y plantillas, pero aún sin un modelo definido.
  - El resultado debe ser algo así:



EN RESUMEN

- El **Patrón MVT de Django** (variante del MVC) facilita mucho el trabajo con los diferentes elementos del lenguaje, ya que permite **abstraerse** tanto de los datos como de la comunicación con el cliente web.
- Acceder a los parámetros enviados por un formulario se facilita mucho con el objeto **`request`**.
- Además, para enviar la información al cliente, **las plantillas** y la **función `render()`** dejan el trabajo mucho más sencillo.
- Con esto, armar una app web se transforma en algo **rápido y de poco esfuerzo**.



Crear bases de datos, migraciones, manejo de usuarios, panel de admin, sistema de permisos...

Express

django

UNIVERSIDAD TECNOLÓGICA DE CHILE
INSTITUTO PROFESIONAL
CENTRO DE FORMACIÓN TÉCNICA