Python: Django Framework

Posma Group

Contenido

1. Introducción
2. Vistas basadas en clases
3. Formularios
4. Sistema de plantillas
5. Managers

Introducción

En la clase anterior nos dimos a conocer con Django, el cual es un framework de amplio espectro de funcionalidades para el desarrollo web en Python. En esta oportunidad estudiaremos con más profundidad las características y herramientas del framework Django.

Vistas basadas en clases

Hasta ahora hemos definido las vistas como funciones. Ésta, sin embargo, no es la única manera de definir vistas. Una manera alternativa es definiéndolas como clases.

Vistas basadas en clases

Hasta ahora hemos definido las vistas como funciones. Ésta, sin embargo, no es la única manera de definir vistas. Una manera alternativa es definiéndolas como clases.

Esta forma de definir vistas provee una serie de ventajas:

- Mejor organización del código respecto a los diferentes métodos HTTP (GET, POST, etc)

Vistas basadas en clases

Hasta ahora hemos definido las vistas como funciones. Ésta, sin embargo, no es la única manera de definir vistas. Una manera alternativa es definiéndolas como clases.

Esta forma de definir vistas provee una serie de ventajas:

- Mejor organización del código respecto a los diferentes métodos HTTP (GET, POST, etc)
- Uso de Programación Orientada a Objetos para estructurar y jerarquizar las vistas

Vistas basadas en clases

Hasta ahora hemos definido las vistas como funciones. Ésta, sin embargo, no es la única manera de definir vistas. Una manera alternativa es definiéndolas como clases.

Esta forma de definir vistas provee una serie de ventajas:

- Mejor organización del código respecto a los diferentes métodos HTTP (GET, POST, etc)
- Uso de Programación Orientada a Objetos para estructurar y jerarquizar las vistas
- Uso de vistas genéricas definidas por el framework

Vistas basadas en clases

Para definir el comportamiento de una vista en caso de recibir una solicitud de tipo GET, utilizando funciones, lo haríamos de la siguiente forma:

```
from django.http import HttpResponse

def mi_vista(request):
    if request.method == 'GET':
        # lógica de la vista
        return HttpResponse('result')
```

Vistas basadas en clases

La misma vista, pero basada en una clase, la definimos así:

```
from django.http import HttpResponse
from django.views.generic.base import View

class MiVista(View):
    def get(self, request):
        # lógica de la vista
        return HttpResponse('result')
```

Vistas basadas en clases

La misma vista, pero basada en una clase, la definimos así:

```
from django.http import HttpResponse
from django.views.generic.base import View

class MiVista(View):
    def get(self, request):
        # lógica de la vista
        return HttpResponse('result')
```

Una vista basada en clase actúa de la misma forma que una vista normal, con la particularidad de que permite una mejor manera de organización.

Vistas basadas en clases

En los archivos de definición de patrones de URL, se espera corresponder con vistas como funciones. Para esto se utiliza el método `as_view()`:

```
# urls.py
from django.conf.urls import patterns
from myapp.views import MyView

urlpatterns = patterns('',
    (r'^ayuda/', AyudaView.as_view()),
)
```

Vistas basadas en clases

En los archivos de definición de patrones de URL, se espera corresponder con vistas como funciones. Para esto se utiliza el método `as_view()`:

```
# urls.py
from django.conf.urls import patterns
from myapp.views import MyView

urlpatterns = patterns('',
    (r'^ayuda/', AyudaView.as_view()),
)
```

El método `as_view` también puede recibir cualquier cantidad de argumentos. Éstos sobrescribirán cualquier atributo declarado en la clase de dicha vista.

Vistas genéricas

Una de las principales ventajas de utilizar clases para definir las vistas, es que tenemos acceso a un considerable número de vistas genéricas definidas por defecto en el framework.

Vistas genéricas

Una de las principales ventajas de utilizar clases para definir las vistas, es que tenemos acceso a un considerable número de vistas genéricas definidas por defecto en el framework.

Todas las vistas genéricas extienden la clase `View`. Lo primero que una vista realiza es invocar el método `dispatch()`, el cual se encarga de invocar al método apropiado dependiendo del tipo de solicitud HTTP (GET, POST, HEAD, etc).

Vistas genéricas

Una de las principales ventajas de utilizar clases para definir las vistas, es que tenemos acceso a un considerable número de vistas genéricas definidas por defecto en el framework.

Todas las vistas genéricas extienden la clase `View`. Lo primero que una vista realiza es invocar el método `dispatch()`, el cual se encarga de invocar al método apropiado dependiendo del tipo de solicitud HTTP (GET, POST, HEAD, etc).

En caso de recibir una solicitud de un tipo no soportado, invoca a `http_method_not_allowed()`

TemplateView

Esta vista retorna un *render* de una plantilla en particular. Bastante útil cuando lo único que se requiere es mostrar un template.

```
from django.views.generic.base import TemplateView

class HomePageView(TemplateView):

    template_name = "home.html"
```

TemplateView

Esta vista retorna un *render* de una plantilla en particular. Bastante útil cuando lo único que se requiere es mostrar un template.

```
from django.views.generic.base import TemplateView

class HomePageView(TemplateView):

    template_name = "home.html"
```

El orden de procesamiento de TemplateView es el siguiente:

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_context_data()`

TemplateView

Si queremos pasar variables al contexto de la plantilla, sobrescribimos `get_context_data`:

```
from django.views.generic.base import TemplateView
from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super(HomePageView, self).get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context
```

RedirectView

Como su nombre lo indica, simplemente efectúa una redirección a un URL dado.

```
from django.views.generic.base import RedirectView

class GoogleRedirectView(RedirectView):

    url = "https://www.google.com/"
```

RedirectView

Como su nombre lo indica, simplemente efectúa una redirección a un URL dado.

```
from django.views.generic.base import RedirectView

class GoogleRedirectView(RedirectView):

    url = "https://www.google.com/"
```

Sin embargo, no es necesario definir una subclase:

```
from django.views.generic.base import RedirectView

urlpatterns = patterns('',
    url(r'^django-doc/$', RedirectView.as_view(url='http://djangoproject.com')),
)
```

RedirectView

El flujo de ejecución de `RedirectView` ocurre de la siguiente forma:

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_redirect_url()`

RedirectView

El flujo de ejecución de `RedirectView` ocurre de la siguiente forma:

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_redirect_url()`

Si la construcción del URL requiere algún tipo de procesamiento, se sobrescribe `get_redirect_url`:

RedirectView

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from libros.models import Libro


class GoogleBookInfoRedirectView(RedirectView):

    url = "https://www.google.com/search"
    query_string = True

    def get_redirect_url(self, *args, **kwargs):
        libro = get_object_or_404(Libro, pk=int(kwargs['pk']))
        self.url +=  "?q=%s" % libro.titulo
        return super(GoogleBookInfoRedirectView, self).get_redirect_url(*args,
                **kwargs)
```

DetailView

Esta vista se utiliza para obtener información de detalle sobre un objeto en particular, especificando como atributo de la vista el modelo asociado.

DetailView

Esta vista se utiliza para obtener información de detalle sobre un objeto en particular, especificando como atributo de la vista el modelo asociado.

Por ejemplo, podemos volver a escribir nuestra vista de detalle de autor utilizando vistas génericas:

```
class AutorDetailView(DetailView):

    model = Autor
    template_name = "libros/autor_detail.html"
    context_object_name = "autor"
```

DetailView

Esta vista se utiliza para obtener información de detalle sobre un objeto en particular, especificando como atributo de la vista el modelo asociado.

Por ejemplo, podemos volver a escribir nuestra vista de detalle de autor utilizando vistas génericas:

```
class AutorDetailView(DetailView):

    model = Autor
    template_name = "libros/autor_detail.html"
    context_object_name = "autor"
```

Ahora agregamos esta línea a `urls.py`:

```
url(r'^autor/(?P<pk>\d+)/$', AutorDetailView.as_view(), name='autor_detail'),
```

DetailView

El flujo de llamadas en un `DetailView` ocurre en el siguiente orden:

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_template_names()`
4. `get_slug_field()`
5. `get_queryset()`
6. `get_object()`
7. `get_context_object_name()`
8. `get_context_data()`
9. `get()`
10. `render_to_response()`

ListView

Se utiliza para mostrar listados de objetos de algún modelo.

ListView

Se utiliza para mostrar listados de objetos de algún modelo.

Podemos sobrescribir nuestra vista `libros.views.index` utilizando un `ListView` en su lugar:

```
class AutorListView(ListView):

    model = Autor
    template_name = "libros/index.html"
    context_object_name = "autores_list"
```

ListView

Se utiliza para mostrar listados de objetos de algún modelo.

Podemos sobrescribir nuestra vista `libros.views.index` utilizando un `ListView` en su lugar:

```
class AutorListView(ListView):

    model = Autor
    template_name = "libros/index.html"
    context_object_name = "autores_list"
```

Si hacemos los cambios necesarios al archivo de URLs, deberíamos preservar el funcionamiento del sitio, ahora con un código más elegante.

Vistas genéricas

ListView

La secuencia de llamadas que Listview ejecuta es la siguiente:

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_template_names()`
4. `get_queryset()`
5. `get_context_object_name()`
6. `get_context_data()`
7. `get()`
8. `render_to_response()`

Formularios

Es posible procesar formularios procesando manualmente los objetos `HttpRequest`, pero Django cuenta con la biblioteca `django.forms` para el manejo de formularios.

Formularios

Es posible procesar formularios procesando manualmente los objetos `HttpRequest`, pero Django cuenta con la biblioteca `django.forms` para el manejo de formularios.

A través de la biblioteca de manejo de formularios, es posible:

- Desplegar un formulario HTML con *widgets* generados automáticamente

Formularios

Es posible procesar formularios procesando manualmente los objetos `HttpRequest`, pero Django cuenta con la biblioteca `django.forms` para el manejo de formularios.

A través de la biblioteca de manejo de formularios, es posible:

- Desplegar un formulario HTML con *widgets* generados automáticamente
- Validar automáticamente los datos introducidos en un formulario

Formularios

Es posible procesar formularios procesando manualmente los objetos `HttpRequest`, pero Django cuenta con la biblioteca `django.forms` para el manejo de formularios.

A través de la biblioteca de manejo de formularios, es posible:

- Desplegar un formulario HTML con *widgets* generados automáticamente
- Validar automáticamente los datos introducidos en un formulario
- Convertir los datos introducidos a sus respectivos tipos de datos en Python

Formularios

La clase Form

Un objeto de tipo `Form` encapsula una secuencia de campos y un conjunto de reglas de validación que deben cumplirse para que el formulario sea aceptado.

Formularios

La clase Form

Un objeto de tipo `Form` encapsula una secuencia de campos y un conjunto de reglas de validación que deben cumplirse para que el formulario sea aceptado.

Podemos utilizar un `Form` para implementar una funcionalidad de "Contáctanos", de la siguiente forma:

```
from django import forms

class ContactanosForm(forms.Form):
    asunto = forms.CharField(max_length=100)
    mensaje = forms.CharField()
    remitente = forms.EmailField()
    reenviar_remitente = forms.BooleanField(required=False)
```

La clase Form

Un objeto de la clase `Form` puede encontrarse asociado o no a un conjunto de datos. Un formulario con datos (*bound*) es capaz de hacer validaciones y mostrar un render.[en] HTML con sus valores correspondientes. Un formulario sin datos (*unbound*) únicamente puede mostrar en HTML sus campos vacíos.

La clase Form

Un objeto de la clase `Form` puede encontrarse asociado o no a un conjunto de datos. Un formulario con datos (*bound*) es capaz de hacer validaciones y mostrar un render.[en] HTML con sus valores correspondientes. Un formulario sin datos (*unbound*) únicamente puede mostrar en HTML sus campos vacíos.

Para crear un formulario, se instancia la clase correspondiente:

```
>>> f = ContactanosForm()
```

La clase Form

Para asociar datos a un formulario, se le debe pasar un diccionario como primer argumento al constructor de la clase:

```
>>> data = {'asunto': 'Hola',
...         'mensaje': 'Hola, cómo estás?',
...         'recipiente': 'yo@ejemplo.com',
...         'reenviar_recipiente': True}
>>> f = ContactanosForm(data)
```

Formularios desde la vista

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

def contact(request):
    if request.method == 'POST':
        form = ContactanosForm(request.POST)
        if form.is_valid():
            # Procesar datos
            return HttpResponseRedirect('/gracias/')
    else:
        form = ContactanosForm()

    return render(request, 'contactanos.html', {
        'form': form,
    })
```

Formularios desde la vista

Utilizando vistas basadas en clase, lo haríamos de esta forma:

```
class ContactView(View):
    form_class = ContactanosForm
    template_name = 'contactanos.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class()
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # procesar datos
            return HttpResponseRedirect('/gracias/')

        return render(request, self.template_name, {'form': form})
```

Formularios desde la vista

Lo anterior puede implementarse de una manera todavía más sencilla, utilizando la vista genérica `FormView`:

Formularios desde la vista

Lo anterior puede implementarse de una manera todavía más sencilla, utilizando la vista genérica `FormView`:

```
from django.views.generic.base import FormView
from .forms import ContactanosForm

class ContactView(FormView):
    form_class = ContactanosForm
    template_name = 'contactanos.html'
    success_url = '/gracias/'

    def form_valid(self, form):
        # procesar datos
        return super(ContactView, self).form_valid(form)
```

Formularios desde el template

Del lado de las plantillas simplemente mostramos el objeto con nombre form que está siendo pasado desde la vista a través del contexto:

```
<form action="/contact/" method="post">
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

Formularios desde el template

Del lado de las plantillas simplemente mostramos el objeto con nombre form que está siendo pasado desde la vista a través del contexto:

```
<form action="/contact/" method="post">
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

En el ejemplo estamos usando el método `as_p` para desplegar el formulario como una serie de etiquetas <p> en HTML. También es posible mostrar el formulario invocando los métodos `as_ul()` para una lista sin orden y `as_table()` para mostrarlo como una tabla.

El API de la clase Form

La clase form cuenta con los siguientes métodos:

is_bound()

```
>>> f = ContactanosForm()
>>> f.is_bound
False
>>> f = ContactanosForm({'asunto': 'hola'})
>>> f.is_bound
True
```

El API de la clase Form

is_valid()

La principal funcionalidad de los formularios es la validación automática de los datos.

```
>>> data = {'asunto': 'Hola',
...         'mensaje': 'Hola, todo bien?',
...         'recipiente': 'yo@ejemplo.com',
...         'reenviar_recipiente': True}
>>> f = ContactanosForm(data)
>>> f.is_valid()
True
```

El API de la clase Form

is_valid()

Si intentamos introducir un dato inválido u omitir un campo requerido, el método retorna `False`. Por defecto, todos los campos de un formulario se asumen como requeridos.

```
>>> data = {'asunto': '',
...         'mensaje': 'Hola, todo bien?',
...         'recipiente': 'yo@ejemplo.com',
...         'reenviar_recipiente': True}
>>> f = ContactanosForm(data)
>>> f.is_valid()
False
```

El API de la clase Form

errors

Todo objeto de tipo `Form` tiene un atributo `errors`, en donde se obtiene un diccionario con los errores ocurridos durante la validación.

```
>>> f.errors
{'asunto': [u'This field is required.']}
```

El API de la clase Form

`fields`

A través del atributo fields, el formulario guarda un diccionario con todos sus campos. En donde cada clave será el nombre del campo, y el valor será el objeto correspondiente de tipo `Field`

```
>>> for campo in f.fields.values():
...     print campo
...
<django.forms.fields.CharField object at 0x24bb510>
<django.forms.fields.CharField object at 0x24bb5d0>
<django.forms.fields.EmailField object at 0x24bb650>
<django.forms.fields.BooleanField object at 0x24bb6d0>
```

El API de la clase Form

`fields`

A través del atributo fields, el formulario guarda un diccionario con todos sus campos. En donde cada clave será el nombre del campo, y el valor será el objeto correspondiente de tipo `Field`

```
>>> for campo in f.fields.values():
...     print campo
...
<django.forms.fields.CharField object at 0x24bb510>
<django.forms.fields.CharField object at 0x24bb5d0>
<django.forms.fields.EmailField object at 0x24bb650>
<django.forms.fields.BooleanField object at 0x24bb6d0>
```

Ahora intentemos explorar usando `dir()` y `help()` en esos campos y sus atributos.

El API de la clase Form

`fields`

Cada campo también tiene un atributo `errors`, así como un atributo `label` y `label_tag`. Esto es útil si queremos desplegar el formulario de forma personalizada, sin estar restringido a los métodos `as_p`, `as_ul`, etc.

El API de la clase Form

fields

Cada campo también tiene un atributo `errors`, así como un atributo `label` y `label_tag`. Esto es útil si queremos desplegar el formulario de forma personalizada, sin estar restringido a los métodos `as_p`, `as_ul`, etc.

```
<form action="/contactanos/" method="post">
    {% for field in form %}
        <div class="fieldWrapper">
            {{ field.errors }}
            {{ field.label_tag }} {{ field }}
        </div>
    {% endfor %}
    <p><input type="submit" value="Enviar" /></p>
</form>
```

El API de la clase Form

`cleaned_data`

Cada campo en un formulario no sólo es responsable de validar sus valores, sino además de normalizarlos en un formato consistente. Por ejemplo, un campo de tipo `DateField` se convierte a un objeto Python de tipo `datetime.date`.

El API de la clase Form

cleaned_data

Cada campo en un formulario no sólo es responsable de validar sus valores, sino además de normalizarlos en un formato consistente. Por ejemplo, un campo de tipo `DateField` se convierte a un objeto Python de tipo `datetime.date`.

Una vez que se ha creado un objeto de tipo `Form` y éste ha validado sus datos, es posible acceder al atributo `cleaned_data`:

```
>>> f.is_valid()
True
>>> f.cleaned_data
{'reenviar_remitente': True, 'mensaje': u'Todo bien?',
'remitente': u'yo@ejemplo.com', 'asunto': u'hola'}
```

El API de la clase Form

cleaned_data

Cada campo en un formulario no sólo es responsable de validar sus valores, sino además de normalizarlos en un formato consistente. Por ejemplo, un campo de tipo `DateField` se convierte a un objeto Python de tipo `datetime.date`.

Una vez que se ha creado un objeto de tipo `Form` y éste ha validado sus datos, es posible acceder al atributo `cleaned_data`:

```
>>> f.is_valid()
True
>>> f.cleaned_data
{'reenviar_remitente': True, 'mensaje': u'Todo bien?',
'remitente': u'yo@ejemplo.com', 'asunto': u'hola'}
```

Cada vez que se necesite procesar los datos de un formulario desde una vista, esto debe hacerse accediendo a `cleaned_data`.

El API de la clase Form

error_css_class, required_css_class

A veces es necesario especificar una clase de estilo para los mensajes de error o de campos requeridos:

```
class ContactanosForm(Form):
    error_css_class = 'error'
    required_css_class = 'required'
```

Widgets

Un `widget` implementa la manera de mostrar un elemento de un formulario. Cada vez que se define un campo de un `Form`, éste viene asociado con un widget por defecto según su tipo de dato.

Widgets

Un `widget` implementa la manera de mostrar un elemento de un formulario. Cada vez que se define un campo de un `Form`, éste viene asociado con un widget por defecto según su tipo de dato.

Sin embargo, es posible especificar un widget en particular para algún campo:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

Widgets

Un `widget` implementa la manera de mostrar un elemento de un formulario. Cada vez que se define un campo de un `Form`, éste viene asociado con un widget por defecto según su tipo de dato.

Sin embargo, es posible especificar un widget en particular para algún campo:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

Esto hará que el campo se muestre como un `<textarea>`, en lugar de un `<input type='text'>`, que es el widget por defecto.

Widgets

Todos los widgets heredan de la clase `django.forms.Widget`, y es posible para el usuario implementar sus propios widgets en el caso de ser necesario.

Widgets

Todos los widgets heredan de la clase `django.forms.Widget`, y es posible para el usuario implementar sus propios widgets en el caso de ser necesario.

Django implementa los siguientes widgets básicos:

- `TextInput`
- `NumberInput`
- `EmailInput`
- `URLInput`
- `PasswordInput`
- `HiddenInput`
- `DateInput`
- `DateTimeInput`

Widgets

- `TimeInput`
- `TextArea`
- `CheckboxInput`
- `Select`
- `NullBooleanSelect`
- `SelectMultiple`
- `RadioSelect`
- `FileInput`

Widgets

- `TimeInput`
- `TextArea`
- `CheckboxInput`
- `Select`
- `NullBooleanSelect`
- `SelectMultiple`
- `RadioSelect`
- `FileInput`

...Entre otros.

Formularios en base a modelos

Django implementa la clase `ModelForm`, que se encarga automáticamente de hacer la correspondencia entre el formulario y su modelo respectivo.

Formularios en base a modelos

Django implementa la clase `ModelForm`, que se encarga automáticamente de hacer la correspondencia entre el formulario y su modelo respectivo.

Para definir los formularios de edición correspondientes a nuestros modelos Autor y Libro, crearemos un archivo `forms.py` dentro de la aplicación `libros`:

```
from django.forms import ModelForm
from libros.models import Autor, Libro

class AutorForm(ModelForm):
    class Meta:
        model = Autor

class LibroForm(ModelForm):
    class Meta:
        model = Libro
```

Formularios en base a modelos

Un `ModelForm` asume todos los campos de su modelo correspondiente. Si se quiere especificar un subconjunto de los campos del modelo en formulario, se define el atributo fields dentro de los atributos de `Meta`, de la siguiente forma

Formularios en base a modelos

Un `ModelForm` asume todos los campos de su modelo correspondiente. Si se quiere especificar un subconjunto de los campos del modelo en formulario, se define el atributo fields dentro de los atributos de `Meta`, de la siguiente forma

```
from django.forms import ModelForm
from mi_app.models import Modelo

class MiModeloForm(ModelForm):
    class Meta:
            model = Modelo
            fields ['campo1', 'campo2', 'otro_campo']
```

CreateView

La vista genérica `CreateView` implementa funcionalidades para el manejo de formularios basados en modelos (`ModelForm`).

CreateView

La vista genérica `CreateView` implementa funcionalidades para el manejo de formularios basados en modelos (`ModelForm`).

Vamos a utilizar un `CreateView` para definir una vista de creación de autores. Para esto agregaremos en `views.py`:

```
from django.views.generic.edit import CreateView
from libros.forms import AutorForm

class AutorCreateView(CreateView):
    form_class = AutorForm
    template_name = 'libros/autor_form.html'
    success_url = '/libros/'
```

CreateView

Ahora sólo resta crear el template `autor_form.html` y mapear los URLs necesarios.

```
<!-- templates/libros/autor_form.html -->
<h2>Agregar Autor</h2>

<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Guardar" />
</form>
```

CreateView

Ahora sólo resta crear el template `autor_form.html` y mapear los URLs necesarios.

```
<!-- templates/libros/autor_form.html -->
<h2>Agregar Autor</h2>

<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Guardar" />
</form>
```

```
# libros/urls.py
url(r'^autor/create/$', AutorCreateView.as_view()),
```

CreateView

Ahora agregaremos un enlace desde el índice para hacer referencia al URL:

```
<!-- templates/libros/index.html -->

<h1>Autores destacados</h1>
{% if autores_list %}
    <ul>
    {% for autor in autores_list %}
        <li><a href="/libros/autor/{{ autor.id }}/">{{ autor.nombre }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No hay autores definidos en el sistema.</p>
{% endif %}

<a href="/libros/autor/create/">Agregar autor</a>
```

UpdateView

`UpdateView` muestra un formulario asociado a un modelo para editar y actualizar los campos de un objeto dado.

UpdateView

`UpdateView` muestra un formulario asociado a un modelo para editar y actualizar los campos de un objeto dado.

Primero haremos la plantilla que mostrará el formulario de edición, y la nombraremos `autor_edit.html`:

```
<h2>Editar Autor</h2>

<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Guardar" />
</form>
```

UpdateView

Ahora agreguemos la siguiente vista al archivo `libros/views.py`:

```
class AutorUpdateView(UpdateView):

    model = Autor
    template_name = 'libros/autor_edit.html'
    success_url = '/libros/'

    def form_valid(self, form):
        self.success_url = '/libros/autor/%s/' % self.get_object().id
        return super(AutorUpdateView, self).form_valid(form)
```

UpdateView

Agregaremos ahora un enlace desde la plantilla de detalle del autor, con el fin de poder editar su información:

```
<h1>{{ autor.nombre }}</h1>

<a href="/libros/autor/edit/{{ autor.id }}/">Editar información de autor</a>

{% if autor.libro_set.all %}
    Obras escritas:
    <ul>
    {% for libro in autor.libro_set.all %}
        <li>{{ libro.titulo }} ({{ libro.fecha_pub|date:"Y" }}) <a href="/libros/s
    info]<a/></li>
    {% endfor %}
    </ul>
{% else %}
    <p>Este autor no tiene obras asociadas.</p>
{% endif %}

<a href="/libros/">Volver al inicio</a>
```

UpdateView

Ahora el respectivo patrón en el archivo `libros/urls.py`. Es necesario importar la clase `AutorUpdateView`:

```
url(r'^autor/edit/(?P<pk>\d+)/$', AutorUpdateView.as_view()),
```

UpdateView

Ahora el respectivo patrón en el archivo `libros/urls.py`. Es necesario importar la clase `AutorUpdateView`:

```
url(r'^autor/edit/(?P<pk>\d+)/$', AutorUpdateView.as_view()),
```

...Si hemos hecho todo correctamente, tenemos un formulario de edición de autores.

DeleteView

Para finalizar con las vistas genéricas de edición, implementaremos una vista `DeleteView`, que nos permita eliminar al autor desde su vista de edición.

DeleteView

Para finalizar con las vistas genéricas de edición, implementaremos una vista `DeleteView`, que nos permita eliminar al autor desde su vista de edición.

En el archivo `views.py` agregaremos la siguiente vista:

```
class AutorDeleteView(DeleteView):

    model = Autor
    success_url = '/libros/'
```

DeleteView

Para finalizar con las vistas genéricas de edición, implementaremos una vista `DeleteView`, que nos permita eliminar al autor desde su vista de edición.

En el archivo `views.py` agregaremos la siguiente vista:

```
class AutorDeleteView(DeleteView):

    model = Autor
    success_url = '/libros/'
```

Y agregaremos la siguiente línea a la lista de patrones en `urls.py`:

```
url(r'^autor/delete/(?P<pk>\d+)/$', AutorDeleteView.as_view()),
```

DeleteView

Pondremos un enlace al final de `autor_detail.html` para eliminar al autor:

```
...

<p><a href="/libros/autor/delete/{{ autor.id  }}/">[Eliminar autor]</a></p>

<p><a href="/libros/">Volver al inicio</a></p>
```

DeleteView

Pondremos un enlace al final de `autor_detail.html` para eliminar al autor:

```
...

<p><a href="/libros/autor/delete/{{ autor.id  }}/">[Eliminar autor]</a></p>

<p><a href="/libros/">Volver al inicio</a></p>
```

Y finalmente crearemos un nuevo template llamado `autor_confirm_delete.html` con el siguiente código:

```
<form action="" method="post">{% csrf_token %}
    <p>Está seguro de que desea eliminar a "{{ autor.nombre }}"?</p>
    <input type="submit" value="Sí" />
</form>

<a href="/libros/">Volver al inicio</a>
```

Iteradores

Ejercicio Práctico

Implementar lo necesario para poder agregar, editar y eliminar **libros** desde la plantilla de detalle del autor.

Templates

Un *template* es sencillamente cualquier archivo de texto como HTML, XML, CSV, etc. El cual permite la inclusión de ciertas etiquetas y acceso a variables.

Templates

Un *template* es sencillamente cualquier archivo de texto como HTML, XML, CSV, etc. El cual permite la inclusión de ciertas etiquetas y acceso a variables.

```
{% block titulo %}{{ seccion.titulo }}{% endblock %}

{% block contenido %}
<h1>{{ seccion.titulo }}</h1>

{% for articulo in lista_articulo %}
    <h2>
    <a href="{{ articulo.get_url }}">
        {{ articulo.headline|upper }}
    </a>
    </h2>

    <p>{{ articulo.preview|truncatewords:"100" }}</p>
{% endfor %}

{% endblock %}
```

Templates

Variables

- Las variables se denotan de la forma: {{ `variable` }}.

Templates

Variables

- Las variables se denotan de la forma: {{ `variable` }}.
- Cuando el motor de plantillas encuentra una variable, evalúa esa variable y la reemplaza por su resultado.

Templates

Variables

- Las variables se denotan de la forma: {{ `variable` }}.
- Cuando el motor de plantillas encuentra una variable, evalúa esa variable y la reemplaza por su resultado.
- Los nombres de variables consisten en cualquier combinación de caracteres alfanuméricos y underscore ("_")

Templates

Variables

- Las variables se denotan de la forma: {{ `variable` }}.
- Cuando el motor de plantillas encuentra una variable, evalúa esa variable y la reemplaza por su resultado.
- Los nombres de variables consisten en cualquier combinación de caracteres alfanuméricos y underscore (“_”)
- Puede accederse a cualquier atributo de la variable utilizando el punto (“.”)

Templates

Variables

- Las variables se denotan de la forma: {{ variable }}.
- Cuando el motor de plantillas encuentra una variable, evalúa esa variable y la reemplaza por su resultado.
- Los nombres de variables consisten en cualquier combinación de caracteres alfanuméricos y underscore ("_")
- Puede accederse a cualquier atributo de la variable utilizando el punto (".")

```
<ul>
  {% for articulo in lista_articulo %}
      <li>{{ articulo.titulo }}</li>
  {% endfor %}
</ul>
```

Templates

La invocación de métodos se hace sin paréntesis, como si fuese un atributo, y los condicionales y ciclos no están precedidos de dos puntos ":".

```
{% if autor.libro_set.all %}
    Obras escritas:
    <ul>
    {% for libro in autor.libro_set.all %}
        <li>{{ libro.titulo }} <a href="/libros/search/{{ libro.id }}/"></li>
    {% endfor %}
    </ul>
{% else %}
    <p>Este autor no tiene obras asociadas.</p>
{% endif %}
```

Filtros

Los filtros se aplican para modificar el valor de una variable antes de mostrarlo, y lucen de la forma:

```
{{ nombre|lower }}
```

Filtros

Los filtros se aplican para modificar el valor de una variable antes de mostrarlo, y lucen de la forma:

```
{{ nombre|lower }}
```

En este caso, todos los caracteres de la cadena `nombre` serán convertidos a minúscula. Django implementa una gran cantidad de filtros nativos. Estudiaremos algunos...

Filtros

add

Suma una valor adicional al existente:

```
{{ cantidad|add:"2" }}
```

Filtros

add

Suma una valor adicional al existente:

```
{{ cantidad|add:"2" }}
```

addslashes

"Escapa" los caracteres de comilla antes de ponerlos en el template:

```
{{ cadena|addslashes }}
```

Filtros

add

Suma una valor adicional al existente:

```
{{ cantidad|add:"2" }}
```

addslashes

"Escapa" los caracteres de comilla antes de ponerlos en el template:

```
{{ cadena|addslashes }}
```

date

Muestra una variable de tipo fecha dado un formato específico:

```
{{ fecha|date:"D d M Y" }}
```

Filtros

default

Si una variable es `False` o vacía, se usa la expresión dada por defecto:

```
{{ email|default:"Ingrese su email" }}
```

Filtros

default

Si una variable es `False` o vacía, se usa la expresión dada por defecto:

```
{{ email|default:"Ingrese su email" }}
```

length

```
{{ autor.libro_set.all|length }}
```

Filtros

default

Si una variable es `False` o vacía, se usa la expresión dada por defecto:

```
{{ email|default:"Ingrese su email" }}
```

length

```
{{ autor.libro_set.all|length }}
```

striptags

Suprime todas las etiquetas HTML o XML de una cadena de texto.

```
{{ valor|striptags }}
```

Filtros

random

Retorna un elemento aleatorio de una colección:

```
{{ lista|random }}
```

Filtros

random

Retorna un elemento aleatorio de una colección:

```
{{ lista|random }}
```

Django implementa muchísimos más filtros por defecto, y es posible para el usuario definir sus propios filtros. Para una información más detallada, referirse a la documentación oficial de django (https://www.djangoproject.com/).

Template tags

Un *template tag* se define como "{% tag %}", y definen comportamientos complejos del sistema de templates, como los ciclos y condicionales que hemos visto hasta ahora.

Template tags

Un *template tag* se define como "{ % tag % }", y definen comportamientos complejos del sistema de templates, como los ciclos y condicionales que hemos visto hasta ahora.

Adicionales a los condicionales y ciclos, estudiaremos algunas de las etiquetas más importantes.

Template tags

{% comment %}

Comenta un segmento del texto:

```
<p>Publicado el {{ pub_date|date:"DD-MM-YY" }}</p>
{% comment %}
    <p>eliminar <input type="button"></p>
{% endcomment %}
```

Template tags

{% comment %}

Comenta un segmento del texto:

```
<p>Publicado el {{ pub_date|date:"DD-MM-YY" }}</p>
{% comment %}
    <p>eliminar <input type="button"></p>
{% endcomment %}
```

{% csrf_token %}

Esta etiqueta existe para proteger el sitio de ataques de CSRF. Se coloca en los formularios que utilicen método POST.

```
<form action="." method="post">{% csrf_token %}
```

Template tags

{% cycle %}

`cycle` va retornando secuencialmente los elementos de una tupla cada vez que la etiqueta aparece. Esto es muy útil para alternar entre valores para modificar el template:

```
{% for obj in una_lista %}
    <tr class="{% cycle 'dark' 'light' %}">
        ...
    </tr>
{% endfor %}
```

Template tags

{% autoescape %}

Controla la interpretación de caracteres HTML dentro de una cadena de texto. Si la opción está en `on`, las etiquetas HTML no serán interpretadas como tal.

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

Template tags

{% autoescape %}

Controla la interpretación de caracteres HTML dentro de una cadena de texto. Si la opción está en `on`, las etiquetas HTML no serán interpretadas como tal.

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

{% url %}

Devuelve un URL dado una vista definida en algún archivo de URLs.

```
{% url 'nombre_de_una_vista' arg1=v1 arg2=v2 %}
```

Herencia de Templates

Una de las particularidades del sistema de plantillas de Django, es que éstas pueden diseñarse jerárquicamente, a través de la herencia y sobrescritura de templates.

Herencia de Templates

Una de las particularidades del sistema de plantillas de Django, es que éstas pueden diseñarse jerárquicamente, a través de la herencia y sobrescritura de templates.

Usualmente se define un template base que contenga la estructura más externa del HTML, y que se encargue de incluir todos los archivos de estilo y javascript necesarios globalmente. Y entonces los demás templates heredan de éste.

Herencia de Templates

Una de las particularidades del sistema de plantillas de Django, es que éstas pueden diseñarse jerárquicamente, a través de la herencia y sobrescritura de templates.

Usualmente se define un template base que contenga la estructura más externa del HTML, y que se encargue de incluir todos los archivos de estilo y javascript necesarios globalmente. Y entonces los demás templates heredan de éste.

Creemos un archivo llamado “base.html” con el siguiente código:

Herencia de Templates

```
<!DOCTYPE html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block titulo %}Bienvenidos a libronline{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block menu %}
        <ul>
            <li><a href="/libros/">Índice de autores</a></li>
            <li><a href="#">Contactenos</a></li>
        </ul>
        {% endblock %}
    </div>

    <h1>Libronline, tu red de lectura</h1>
    <div id="content">
    {% block contenido %}{% endblock %}
    </div>
</body>
</html>
```

Herencia de Templates

Y ahora sobrescribiremos el template que está en `libros/index.html` para que herede de éste.

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

{% block titulo %}Bienvenidos a libronline - Autores {% endblock %}

{% block contenido %}
<h1>Autores destacados</h1>
{% if autores_list %}
    <ul>
    {% for autor in autores_list %}
        <li><a href="/libros/autor/{{ autor.id }}/">{{ autor.nombre }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No hay autores definidos en el sistema.</p>
{% endif %}

<a href="/libros/autor/create/">Agregar autor</a>
{% endblock %}
```

Herencia de Templates

De esta forma cada template derivado de `base.html`, se encarga de sobrescribir los bloques que necesite, refiriéndose a ellos por sus mismos nombres en el template base.

Herencia de Templates

De esta forma cada template derivado de `base.html`, se encarga de sobrescribir los bloques que necesite, refiriéndose a ellos por sus mismos nombres en el template base.

La herencia de templates permite una estructuración ordenada y funcional de las plantillas en el proyecto, promoviendo el reciclaje y la legibilidad del código HTML.

Managers

Todos los modelos de Django incluyen un objeto `Manager` que se encargará de implementar las conexiones a la fuente de datos.

Managers

Todos los modelos de Django incluyen un objeto `Manager` que se encargará de implementar las conexiones a la fuente de datos.

Por defecto, cada modelo tiene un atributo llamado `objects`, el cual es un objeto de tipo `Manager`. Pero podríamos querer que el manejador se llame de otra forma, o implementar varios objetos `Manager` dentro del mismo modelo.

Managers

Todos los modelos de Django incluyen un objeto `Manager` que se encargará de implementar las conexiones a la fuente de datos.

Por defecto, cada modelo tiene un atributo llamado `objects`, el cual es un objeto de tipo `Manager`. Pero podríamos querer que el manejador se llame de otra forma, o implementar varios objetos `Manager` dentro del mismo modelo.

```
from django.db import models

class Persona(models.Model):
    #...
    gente = models.Manager()
```

Managers

Es posible extender la clase `Manager` creando nuevos manejadores. Por ejemplo, podemos definir un `AcademicBookManager`, que retorne los libros de una vez filtrando por género:

```
from django.db import models

class AcademicBookManager(models.Manager):

    def get_queryset(self):
        return super(AcademicBookManager, self).get_queryset().filter(genero=4)
```

Managers

Agregamos ahora los manejadores al comienzo de la clase `Libro`:

```
class Libro(models.Model):

    objects = models.Manager()
    academic = AcademicBookManager()
    # ...
```

Managers

Agregamos ahora los manejadores al comienzo de la clase `Libro`:

```
class Libro(models.Model):

    objects = models.Manager()
    academic = AcademicBookManager()
    # ...
```

Ahora tenemos nuestro modelo con su manejador `objects` como es lo normal, pero adicionalmente podemos usar el otro `Manager`:

```
Libro.academic.all()
```

Manejo de SQL

Otra característica útil de definir nuestros propios manejadores de modelos, es la posibilidad de lidiar directamente con la base de datos.

Manejo de SQL

Otra característica útil de definir nuestros propios manejadores de modelos, es la posibilidad de lidiar directamente con la base de datos.

La clase Manager cuenta con el método `raw()` para ejecutar *queries* directamente:

```
>>> from libros.models import Autor
>>> for a in Autor.objects.raw('Select * from libros_autor'):
...     print a
...
Herman Hesse
Paulo Coelho
Serafin Mazparrote
Milan Kundera
```

Manejo de SQL

Es posible pasar cualquier cantidad de parámetros a la función `raw()` como una lista o un diccionario de argumentos:

```
>>> apellido = 'Perez'
>>> Persona.objects.raw('SELECT * FROM myapp_persona WHERE last_name = %s',
...     [apellido])
```

Manejo de SQL

Es posible pasar cualquier cantidad de parámetros a la función `raw()` como una lista o un diccionario de argumentos:

```
>>> apellido = 'Perez'
>>> Persona.objects.raw('SELECT * FROM myapp_persona WHERE last_name = %s',
...     [apellido])
```

Este mecanismo protege las consultas a base de datos de posibles ataques de SQL injection. No se recomienda formatear directamente el texto del query con los parámetros.

Manejo de SQL

A veces es necesario efectuar instrucciones en SQL sin atarlas necesariamente a un modelo. A través del objeto `django.db.connection` pueden efectuarse muchas operaciones directas con la base de datos.

Manejo de SQL

A veces es necesario efectuar instrucciones en SQL sin atarlas necesariamente a un modelo. A través del objeto `django.db.connection` pueden efectuarse muchas operaciones directas con la base de datos.

```
from django.db import connection

def mi_funcion_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])

    row = cursor.fetchone()
    return row
```

Manejo de SQL

A veces es necesario efectuar instrucciones en SQL sin atarlas necesariamente a un modelo. A través del objeto `django.db.connection` pueden efectuarse muchas operaciones directas con la base de datos.

```
from django.db import connection

def mi_funcion_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])

    row = cursor.fetchone()
    return row
```

También existe el método `cursor.fetchall()` para obtener todas las filas de una consulta con varios resultados.

Manejo de SQL

Es posible utilizar un cursor como un manejador de contexto, de la siguiente forma:

```
with connection.cursor() as c:
    c.execute(...)
```

Manejo de SQL

Es posible utilizar un cursor como un manejador de contexto, de la siguiente forma:

```
with connection.cursor() as c:
    c.execute(...)
```

Esto, como con los archivos, nos permite cierta legibilidad, nos evita la necesidad de cerrarlo, y encapsula secciones del código para evitar efectos secundarios.

Stored Procedures

El objeto cursor, además de execute para hacer las consultas, tiene un método `callproc`, para invocar *stored procedures* en la base de datos:

Stored Procedures

El objeto cursor, además de execute para hacer las consultas, tiene un método `callproc`, para invocar *stored procedures* en la base de datos:

```
from django.db import connection

with connection.cursor() as cur:
    cur.callproc('proc_name')
```

Stored Procedures

Si necesitamos definir un modelo cuya fuente de datos utilice stored procedures, lo más adecuado es definir un `Manager` específico para que implemente las llamadas necesarias:

Stored Procedures

Si necesitamos definir un modelo cuya fuente de datos utilice stored procedures, lo más adecuado es definir un `Manager` específico para que implemente las llamadas necesarias:

```
from django.db import models, connection


class StoredProcedureManager(models.Manager):

    def metodo_consulta(self, arg1, arg2):
        PROC_NAME = 'consulta_proc'

        cur = connection.cursor()
        cur.callproc(PROC_NAME, [arg1, arg2])
        result = cur.fetchall()
        cur.close()
        return result
```

Conclusión

Django es un framework bastante extenso, por lo cual es difícil cubrir todas sus funcionalidades desde un principio. Sin embargo, hemos cubierto lo suficiente para tener una concepción integral acerca de las herramientas que este framework provee, así como la posibilidad de modificar y extender dichas herramientas.

Muchas gracias!!