HINTS

CRUD EN DJANGO

Django es un marco web basado en Python, el cual permite crear rápidamente aplicaciones web. Tiene una interfaz de administración incorporada que facilita el trabajo con él. A menudo se le llama marco con baterías incluidas, pues proporciona instalaciones integradas para cada funcionalidad. Las vistas genéricas basadas en clases son un conjunto avanzado de vistas integradas que se utilizan para la implementación de estrategias de visualización selectiva como Crear, Recuperar, Actualizar, Eliminar. Las vistas basadas en clases simplifican el uso al separar las solicitudes `GET` y `POST` de una vista. No reemplazan las basadas en funciones, pero tienen ciertas diferencias y ventajas en comparación con ellas:

- La organización del código relacionado con métodos HTTP específicos (`GET`, `POST`, entre otros) se puede abordar mediante métodos separados, en lugar de la ramificación condicional.
- Se pueden utilizar técnicas orientadas a objetos como mixins (herencia múltiple) para factorizar el código en componentes reutilizables.

CRUD GENÉRICO EN DJANGO

Para agilizar nuestros desarrollos se puede crear un CRUD abstracto que se use con cualquiera de los modelos. Esto es muy útil pues no necesitamos hacer las vistas de Create, Remove, Update y Detail para cada uno de nuestros modelos. Partiremos de un estado en el que ya se tienen creados los modelos, que serán: `ModelA`, `ModelB`, y `ModelC`. Utilizaremos las vistas basadas en clases de `'django.views.generic'`, y suponemos que ya tenemos hecha la plantilla base de nuestra aplicación.

Una vez que tenemos definidos nuestros modelo, se les debe dar una funcionalidad extra. Obtener los nombres de los campos que vamos a mostrar en los listados. Generalmente, en los listados de objetos vemos sólo una serie de campos, no todos. Revisaremos un ejemplo: tal como se puede ver, se trata de un método estático, ya que pertenece al modelo y no a una instancia de éste.

```
@staticmethod
def getTableHeaders():
    headers = []

headers.append(ModelA._meta.get_field_by_name('campo_que_quiero_mostrar_1')[0].verbose_name)

headers.append(ModelA._meta.get_field_by_name('campo_que_quiero_mostrar_2')[0].verbose_name)
```

```
    ...
    return headers


```

Obtener los datos para cada objeto que vamos a mostrar. Lo ideal es que se correspondan con los campos que hemos puesto en el método anterior.

```
def getTableData(self):
    data = []
    data.append(self.campo_que_quiero_mostrar_1)
    data.append(self.campo_que_quiero_mostrar_2)
    ...
    return data
```

Obtener las URLs que vamos a necesitar para nuestro CRUD. La obtención de la URL de Create y List, serán métodos estáticos ya que no dependen de ninguna instancia.

```
@staticmethod
def getCreateURL():
    return reverse('ModelACreate')

def getRemoveURL(self):
    return reverse('ModelARemove', kwargs={'pk':self.pk})

def getUpdateURL(self):
    return reverse('ModelAUpdate', kwargs={'pk':self.pk})

def getDetailURL(self):
    return reverse('ModelADetail', kwargs={'pk':self.pk})

@staticmethod
def getListURL():
    return reverse('ModelAList')
```

Obtener el nombre del modelo para saber sobre qué modelo estamos aplicando la vista.

```
class ModelA(models.Model):
        campo_que_quiero_mostrar_1 = models.CharField(max_length=256)
        campo_que_quiero_mostrar_2 = models.CharField(max_length=256)

        class Meta:
                verbose_name = 'Modelo A'
                verbose_name_plural = 'Modelos A'
```

Obtener todos los atributos, y su valor de un objeto.

```
def get_fields(self):
    pairs = []
    for field in self._meta.fields:
        name = field.name
        verbose_name = field.verbose_name
        try:
            pairs.append({'name':name, 'label':verbose_name,
'value':getattr(self, "get_%s_display" % name)()})
        except AttributeError:
            pairs.append({'name':name, 'label':verbose_name,
'value':getattr(self, name)})
    return pairs
```

Ya tenemos unos modelos que son capaces de proporcionarnos toda la información que necesitamos para crear el CRUD.

CREATE

Para casi cualquier vista de creación, lo que necesitamos es una plantilla para pintar el formulario. En este caso, usaremos una sencilla pero que va a servir para todos los modelos. Suponemos que en la plantilla de la que se herede ésta, tendremos un bloque para el contenido, que será el formulario, y la incluiremos en algún sitio al que puedan acceder todas las aplicaciones en `'templates/generic/form.html'`.

```
{% extends 'base.html' %}
{% block content %}
    <h1>{{ title | capfirst }}</h1>
    {{ form }}
{% endblock %}
```

Ahora vamos a hacer una vista que herede de `CreateView`.

```
class genericCreateView(CreateView):
        template_name = 'generic/form.html'

        def get_context_data(self, **kwargs):
                context =
super(genericCreateView,self).get_context_data(**kwargs)
                context['title'] = self.model._meta.verbose_name.title()
                return context

        def get_form(self, form_class):
```

```
                form =
super(genericCreateView,self).get_form(form_class)
                form.helper = FormHelper(form)
                form.helper.layout.append(Submit('accept', 'Aceptar'))
                form.helper.layout.append(HTML('<a href="' +
str(self.model.getListURL()) +'">Cancelar</a>'))
                return form
```

Se le indica que use la plantilla del formulario anterior. En `title` ponemos el nombre que le hemos dado a la clase, y en el formulario le añadimos los botones de aceptar y cancelar. Este último nos llevará otra vez a la lista de todos los objetos.

Por último, asignaremos la URLS. En el ejemplo veremos cómo usar esta vista para todos nuestros modelos.

```
url(r'^modelA/create$',genericCreateView.as_view(model=ModelA,
success_url=reverse_lazy('ModelAList')), name='ModelACreate'),
url(r'^modelB/create$', genericCreateView.as_view(model=ModelB,
success_url=reverse_lazy('ModelBList')), name='ModelBCreate'),
url(r'^modelC/create$', genericCreateView.as_view(model=ModelC,
success_url=reverse_lazy('ModelCList')), name='ModelCCreate'),
```

REMOVE

Esta vista la vamos a usar desde la lista de objetos, por lo que no necesitaremos una plantilla intermedia para pedir la confirmación para eliminar el objeto. La idea es que se puedan acceder a todas las acciones de una forma rápida en la lista de objetos.

```
class genericRemoveView(DeleteView):

    def get(self, request, *args, **kwargs):
        return self.post(request,*args,**kwargs)
```

Como no tenemos formulario intermedio, se pasarán directamente los mismos parámetros del get al post para que se elimine el objeto.

Las URLs.

```
url(r'^modelA/remove/(?P<pk>\d+)$',
genericDeleteView.as_view(model=ModelA,
success_url=reverse_lazy('ModelAList'))), name='ModelARemove'),
url(r'^modelB/remove/(?P<pk>\d+)$',
genericDeleteView.as_view(model=ModelB,
success_url=reverse_lazy('ModelBList'))), name='ModelBRemove'),
url(r'^modelC/remove/(?P<pk>\d+)$',
genericDeleteView.as_view(model=ModelC,
success_url=reverse_lazy('ModelCList'))), name='ModelCRemove'),
```

UPDATE

La vista de Update va a compartir el mismo formulario que la de create, por lo que no vamos a hacer otra plantilla nueva. Serán prácticamente iguales, sólo que en este caso vamos a heredar de UpdateView. Se podría hacer en una única clase, pero el update suele ser más complejo, y se podrían tener que cambiar relaciones y otros elementos, por lo que así se consigue más libertad.

```
class genericUpdateView(UpdateView):
        template_name = 'generic/form.html'

        def get_context_data(self, **kwargs):
                context =
super(genericUpdateView,self).get_context_data(**kwargs)
                context['title'] = self.model._meta.verbose_name.title()
                return context

        def get_form(self, form_class):
                form =
super(genericUpdateView,self).get_form(form_class)
                form.helper = FormHelper(form)
                form.helper.layout.append(Submit('accept', 'Aceptar'))
                form.helper.layout.append(HTML('<a href="' +
str(self.model.getListURL()) +'">Cancelar</a>'))
                return form
```

Y las URLs.

```

url(r'^modelA/update/(?P<pk>\d+)$',
genericDeleteView.as_view(model=ModelA,
success_url=reverse_lazy('ModelAList'))), name='ModelAUpdate'),
url(r'^modelB/update/(?P<pk>\d+)$',
genericDeleteView.as_view(model=ModelB,
success_url=reverse_lazy('ModelBList'))), name='ModelBUpdate'),

```

```
 9 url(r'^modelC/update/(?P<pk>\d+)$',
10 genericDeleteView.as_view(model=ModelC,
11 success_url=reverse_lazy('ModelCList'))), name='ModelCUpdate'),
```

DETAIL

La vista de detalle es muy sencilla y clara. Lo que vamos a hacer es mostrar los campos con sus respectivos valores. A partir de aquí se puede hacer que la vista sea más atractiva, ya que en este caso no va a tener nada de funcionalidad. Volvemos a crear una plantilla que sea accesible desde todas las aplicaciones, por ejemplo en: `'templates/generic/detail.html'`.

```
1 {% extends 'base.html' %}
2 {% block content %}
3     <h1>{{ title | capfirst }}</h1>
4     {% for atrib in object.get_fields %}
5         {{atrib.label}} : {{atrib.value}}
6     {% endfor %}
7 {% endblock %}
```

Esta plantilla la rellenaremos desde la siguiente vista:

```
1 class genericDetailView(DetailView):
2         template_name = 'generic/detail.html'
3
4         def get_context_data(self, **kwargs):
5                 context =
6 super(genericCreateView,self).get_context_data(**kwargs)
7                 context['title'] = self.model._meta.verbose_name.title()
8                 return context
```

Y ahora sólo nos queda añadir las URLs.

```
1 url(r'^modelA/detail/(?P<pk>\d+)$',
2 genericDeleteView.as_view(model=ModelA,
3 success_url=reverse_lazy('ModelAList'))), name='ModelADetail'),
4 url(r'^modelB/detail/(?P<pk>\d+)$',
5 genericDeleteView.as_view(model=ModelB,
6 success_url=reverse_lazy('ModelBList'))), name='ModelBDetail'),
7 url(r'^modelC/detail/(?P<pk>\d+)$',
8 genericDeleteView.as_view(model=ModelC,
9 success_url=reverse_lazy('ModelCList'))), name='ModelCDetail'),
```

LIST

Por último, vamos a agrupar toda la funcionalidad anterior en una vista de lista desde donde podremos administrar todos los objetos. Creamos una plantilla, por ejemplo: **'templates/generic/list.html'**.

```
{% extends 'base.html' %}
{% block content %}
    <h1>{{ title | capfirst }}</h1>
    <a href="{{ createURL }}"><span>+ Nuevo</span></a>
        <table class="table">
                <thead>
                <tr>
                        <th>Opciones</th>
                        {% for header in tableHeaders %}
                                <th>
                                    {{ header }}
                                </th>
                        {% endfor %}
                </tr>
                </thead>
                <tbody>
                {% for object in object_list %}
                        <tr>
                                <td>
                                 <a href="{{ object.getDetailURL
}}"><span>Ver</span></a>
                                 <a href="{{ object.getUpdateURL
}}"><span>Modificar</span></a>
                                 <a href="{{ object.getDeleteURL
}}"><span>Eliminar</span></a>
                                </td>
                                {% for data in object.getTableData %}
                                    <td>
                                        {{ data | default:'-' }}
                                    </td>
                                {% endfor %}
                        </tr>
                {% endfor %}
                </tbody>
        </table>
{% endblock %}
```

Y para generar los datos necesarios:

```
class genericListView(ListView):
        template_name = 'generic/list.html'

        def get_context_data(self, **kwargs):
                context =
super(genericCreateView,self).get_context_data(**kwargs)
```

```
7                context['title'] = self.model._meta.verbose_name.title()
8                context['createURL'] = self.model.getCreateURL()
9                return context
```

Ya sólo nos falta añadir las URLs, y podremos administrar todos nuestros objetos de una forma sencilla, eficaz e intuitiva para los usuarios de nuestra aplicación.

```
1 url(r'^modelA/list$', genericListView.as_view(model=ModelA,
2 success_url=reverse_lazy('ModelAList'))), name='ModelAList'),
3 url(r'^modelB/list$', genericListView.as_view(model=ModelB,
4 success_url=reverse_lazy('ModelBList'))), name='ModelBList'),
5 url(r'^modelC/list$', genericListView.as_view(model=ModelC,
6 success_url=reverse_lazy('ModelCList'))), name='ModelCList'),
```

LIMITACIONES DEL MIDDLEWARE CSRF

`CsrfMiddleware` necesita el framework de sesiones de Django para poder funcionar. Si estás usando un framework de sesiones, o autenticación personalizado, que maneja en forma manual las cookies de sesión, este middleware no funcionará.

Si tu aplicación crea páginas HTML y formularios con algún método inusual (por ejemplo, si envía fragmentos de HTML en sentencias JavaScript `document.write`), podrías estár salteando el filtro que agrega el campo oculto al formulario. De presentarse esta situación, el envío del formulario fallará siempre (esto sucede porque `CsrfMiddleware` usa una expresión regular para agregar el campo `csrfmiddlewaretoken` a tu HTML antes de que la página sea enviada al cliente, y la expresión regular a veces no puede manejar código HTML muy extravagante). Si eso ocurre, sólo examina el código en tu navegador Web para revisar si es que `csrfmiddlewaretoken` ha sido insertado en tu `<form>`.