AWAKELAB

Acceso a Bases de Datos con Python/Django

Implementar una aplicación web MVC que realiza operaciones CRUD en la base de datos utilizando los componentes del framework Django para dar solución a un problema

Objetivo de la jornada

1. Integra la capa de acceso a datos en un aplicativo web MVC para dar solución a un problema
2. Implementa un aplicativo web que realiza operaciones CRUD sobre un modelo utilizando el patrón MVC para dar solución a un problema

CRUD en Django

Qué es CRUD

CRUD significa Crear (Create), Leer (Read), Actualizar (Update) y Eliminar (Delete). Estas son las cuatro operaciones básicas que se ejecutan en modelos de base de datos.

LEER (READ)

La capacidad de la aplicación para leer datos de la base de datos

ACTUALIZAR (UPDATE)

La capacidad de la aplicación para editar el valor almacenado en la base de datos.

ELIMINAR (DELETE)

La capacidad de la aplicación para eliminar el valor en la base de datos.

Vamos a desarrollar las operaciones en el mismo orden.

La mayoría de las aplicaciones en la Web son aplicaciones CRUD. Por ejemplo, Facebook usa operaciones CRUD para guardar sus datos en su base de datos. Puede cambiar su foto de perfil que significa realizar la operación de actualización. Por supuesto, puede ver los datos en la aplicación o el navegador que es la operación de lectura. Además, puede eliminar su cuenta de Facebook, que es la operación de eliminación.

En resumen, casi todas las aplicaciones que utilizamos en la Web son aplicaciones CRUD.

Para los desarrolladores, crear una aplicación CRUD es uno de los primeros pasos. Si podemos crear una aplicación CRUD a partir de nuestro framework, podemos entonces a comenzar la implementación de proyectos.

Gracias a Django contamos con clases específicas las cuales nos ayudarán fácilmente en la creación de un CRUD en nuestro proyecto, sin la necesidad de incorporar lenguaje SQL.

ListView: Esta clase nos permite rescatar un listado de la información que se encuentra en nuestra base de datos, retornando un objeto a nuestro template, para poder iterarlo en nuestro diseño.

```
class NombreClaseVista(ListView):
    model=NombreClaseModel          #Clase creada en model.py para obtener los campos de nuestro formulario
    template_name = "ruta_formulario/diseño_proyecto.html" #ruta del html para visualizar la información
    context_object_name = "ObjetoRetornado" #Objeto retornado para poder iterarlo en nuestro diseño html
```

CreateView: Esta clase nos permite crear o insertar un nuevo valor en la tabla de nuestra base de datos base de datos. Es importante el atributo success_url, ya que le indicamos donde deberá redireccionar, una vez se haya realizado la acción.

```
class NombreClaseCrear(CreateView):
    model=NombreClaseModel          #Clase creada en model.py para obtener los campos de nuestro formulario
    fields='__all__'    #Podemos llamar a una cierta cantidad de campos o por defecto a todos con el valor '__all__'
    success_url=reverse_lazy('ruta_formulario:nombre_en_urls'') #Ruta la cual redireccionará una vez completada
```

DeleteView: Esta clase nos permite eliminar un registro dentro de nuestra tabla en la base de datos.

```
class NombreClaseCrear(DeleteView):
    model=NombreClaseModel          #Clase creada en model.py para obtener los campos de nuestro formulario
    fields='__all__'    #Podemos llamar a una cierta cantidad de campos o por defecto a todos con el valor '__all__'
    success_url=reverse_lazy('ruta_formulario:nombre_en_urls'') #Ruta la cual redireccionará una vez completada
```

UpdateView: Esta clase nos permite actualizar un registro dentro de nuestra tabla en la base de datos.

```
class NombreClaseCrear(UpdateView):
    model=NombreClaseModel          #Clase creada en model.py para obtener los campos de nuestro formulario
    fields='__all__'    #Podemos llamar a una cierta cantidad de campos o por defecto a todos con el valor '__all__'
    success_url=reverse_lazy('ruta_formulario:nombre_en_urls'') #Ruta la cual redireccionará una vez completada
```

Implementación de CRUD en Django

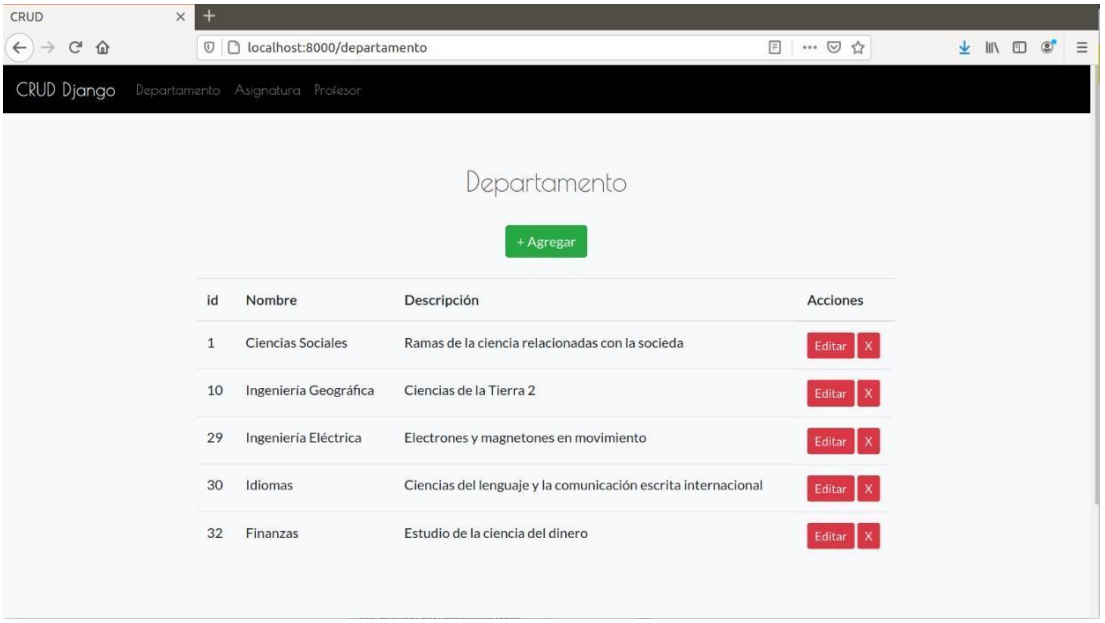
Implementaremos en este apartado las operaciones CRUD para el caso de estudio que hemos venido desarrollando en apartados anteriores sobre Departamentos, Asignaturas y Profesores.

Partimos desde la base de haber dado de alta la aplicación comuna01 e instalado en nuestro archivo **municipio01/settings.py**. En el mismo lugar, nos aseguramos de tener configurada la ruta de archivos estáticos. A continuación el extracto de nuestro archivo **municipio01/settings.py** incluyendo ambas cosas:

```
... INSTALLED_APPS
= [
      'django.contrib.admin',
      'django.contrib.auth',
      'django.contrib.contenttypes',
      'django.contrib.sessions',
      'django.contrib.messages',
      'django.contrib.staticfiles',
      'comuna01'
]
...
STATIC_URL = '/static/'
...
```

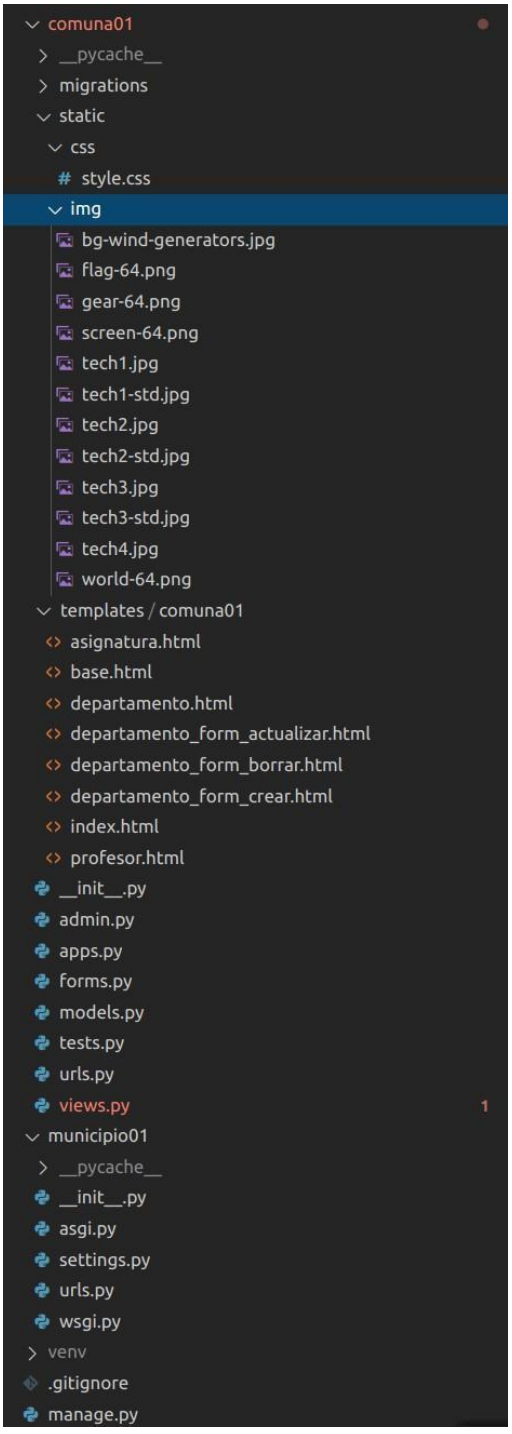
En este caso aprovecharemos algo de lo realizado en el apartado relacionado con **Bootstrap**. Tomaremos como base visual el último ejemplo del documento de clases.

Debemos transportar el código al framework Django. Para esto aplicaremos varios de los aspectos aprendidos hasta el momento, por ejemplo, extensión de Templates, definición de rutas, modelos, formularios y vistas. La intención es construir una aplicación básica que implemente las operaciones CRUD revisadas más arriba, pero en un formato visual que permita a un usuario interactuar con el sistema a través de un front end. El objetivo es lograr algo como lo siguiente:



Con fines didácticos, nos centraremos en el menú **Departamento**. Los casos de **Asignatura** y **Profesor** son implementaciones análogas, cuya implementación se deja como ejercicio al lector.

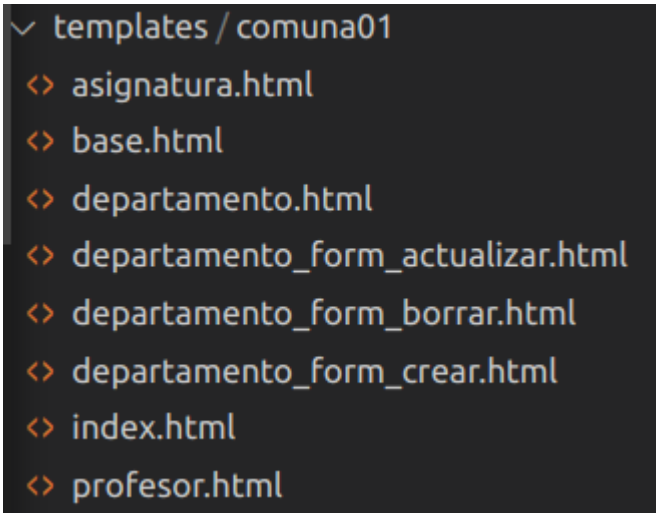
Para lograr lo propuesto debemos intervenir varios archivos de nuestro proyecto, el cual quedará de la siguiente forma:



Puede apreciarse que hemos pasado a **comuna01/templates/** y a **comuna01/static/** todos los archivos **html** y estáticos, como **CSS** e imágenes, para que Django los pueda utilizar al renderizar las páginas según los contenidos y estilos que buscamos lograr.

TEMPLATES

En términos de Templates, nótese que tenemos lo siguiente:



Podemos ver que hemos creado un template **comuna01/base.html**, el que extendemos según las distintas operaciones crud que deseamos realizar:

base.html

```
{% load static %}
<!doctype html>
<html lang="es">
<head>
    <title>CRUD</title>
    <meta charset="utf-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1, shrink-to-fit=no">
<!-- Loading Bootstrap CSS -->
  <link
        rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.c
ss"
        integrity="sha384-
9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk"
        crossorigin="anonymous">
<link href="https://fonts.googleapis.com/css?family=Poiret+One"
      rel="stylesheet" type="text/css">
<link
href="https://fonts.googleapis.com/css?family=Lato&
subset=latin,latin-ext"
      rel="stylesheet" type="text/css">
<link rel="stylesheet" href="{% static 'css/style.css' %}">
{% block scriptsathead %}{% endblock %}
</head>
```

```
<body>

{% block scriptsattop %}{% endblock %}

<!-- Navigation Bar -->
{% block navigationbar %}
    <nav class="navbar navbar-expand-lg navbar-dark fixed-top">
       <a class="navbar-brand" href="/">CRUD Django</a>

       <!-- Navigation Bar Hamburger-->
       <button
               class="navbar-toggler"
               type="button"
               data-toggle="collapse"
               data-target="#navigation"
               aria-expanded="false"
               aria-controls="navigation"
               aria-label="Toggle navigation">
           <span class="navbar-toggler-icon"></span>
       </button>
       < !-- Navigation Links -->
       <div class="collapse navbar-collapse" id="navigation">
           <ul class="navbar-nav mr-auto">
               <li class="nav-item">
                   <a
                           class="nav-link scroll-trigger"
                           href="{% url 'comuna01:url_departamento' %}">
                       Departamento
                   </a>
               </li>
               <li class="nav-item">
                   <a
                      class="nav-link scroll-trigger"
                      href="{% url 'comuna01:url_asignatura'%}">
                        Asignatura
                   </a>
               </li>
               <li class="nav-item">
                   <a
                      class="nav-link scroll-trigger"
                      href="{% url 'comuna01:url_profesor' %}">
                       Profesor
                    </a>
                </li>

           </ul>
       </div>
   </nav>
{% endblock %}

{% block main_content %}

{% endblock %}
```

```
<!--   FOOTER   -->
{#TODO: meter ese padding en CSS file#}
<footer class="fixed-bottom; padding: 100px;">
    <p class="text-muted">&copy; Crud Makers</p>
</footer>
<!-- END FOOTER -->


<!-- jQuery, Popper.js, Bootstrap JS -->

<script
   src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
   integrity="sha384-
DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj"
   crossorigin="anonymous">
</script>
    <script
 src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
      integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
      crossorigin="anonymous">
</script>
   <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"
integrity="sha384-
OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ipu6Tp75j7Bh/kR0JKI"
      crossorigin="anonymous">
</script>
<script type="text/javascript">
    $('.scroll-trigger').click(function () {
        $('.navbar-toggler').click();
    });
</script>
{% block scriptsatbottom %}{% endblock %}
</body>
</html>
```

Con esto podemos extender nuestro estilo de página diseñada en base a `Bootstrap` para las distintas operaciones CRUD de la siguiente forma:

Leer (Read)

comuna01/departamento.html

```
{% extends 'comuna01/base.html' %}

{% block main_content %}
<div class="container-fluid bg-light custom-section"
id="departamento">

<div class="row justify-content-center align-items-center h-100">
    <h2 class="text-center">Departamento</h2>
</div>
```

```
<br>
<div class="row justify-content-center align-items-center h-100">
<a class="btn btn-success" id="add-dpto"
   href="{% url 'comuna01:url_crear_departamento' %}">
       + Agregar
</a>
</div>
<br>
<div class="row justify-content-center align-items-center h-100">
<div class="col col-sm-8 col-md-8 col-lg-8 col-xl-8">
<table class="table table-hover">
<thead>
<tr>
<th>id</th>
<th>Nombre</th>
<th>Descripción</th>
<th>Acciones</th>
</tr>
</thead>
<tbody>
{% for datum in data %}
<tr>
<td>{{ datum.id }}</td>
<td>{{ datum.nombre }}</td>
<td>{{ datum.descripcion }}</td>
<td style="display: inline-block;">
<a class="btn btn-danger btn-sm"
href="{% url 'comuna01:url_actualizar_departamento' pk=datum.id %}">
  Editar
</a>
<a class="btn btn-danger btn-sm" href="{% url
'comuna01:url_borrar_departamento' pk=datum.id %}">X</a></td>
</tr>
{% endfor %}
</tbody>
</table>
</div>
</div>
<div class="row justify-content-center align-items-center h-100">
<div class="col col-sm-8 col-md-8 col-lg-8 col-xl-8">
{% block form_include %}
{% endblock %}
</div>
</div>
</div>
{% endblock %}
```

Lo que logramos en este caso es mostrar la lista de todos los registros de la tabla `departamento` de nuestra base de datos. Tal como se muestra en la última figura mostrada.

Crear (Create)

En este caso necesitamos un formulario que permita al usuario el ingreso de la información para los distintos campos:

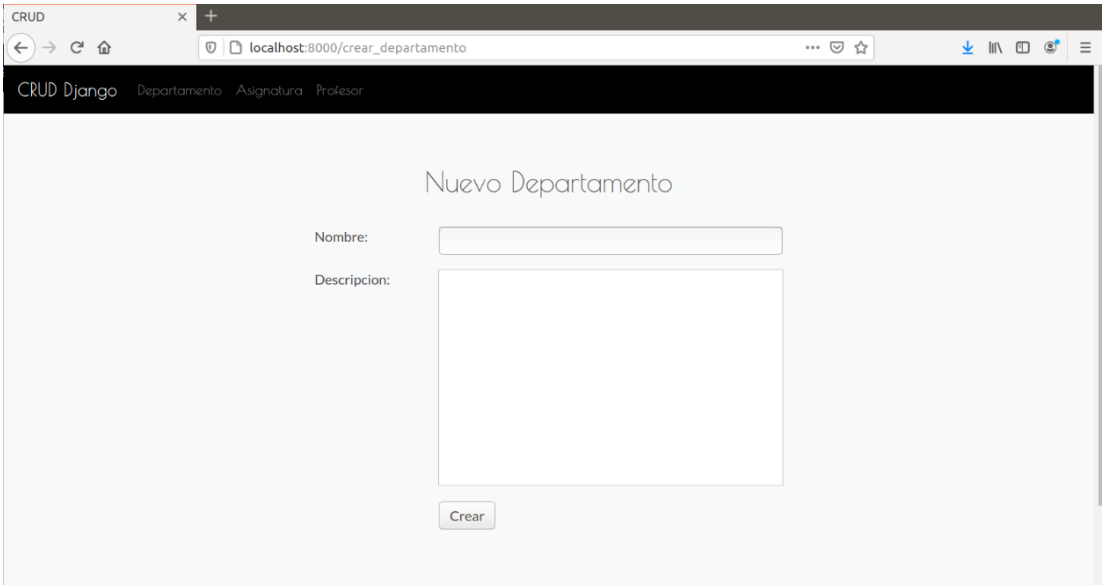
comuna01/departamento_form_crear.html

```
{% extends 'comuna01/base.html' %}

{% block main_content %}
<div class="container-fluid bg-light custom-section crud-form">
  <div class="row justify-content-center align-items-center h-100">
      <h2 class="text-center">Nuevo Departamento</h2>
  </div>
  <br>
  <div class="row justify-content-center align-items-center">
    <form method="post" action="/crear_departamento" novalidate>
    {% csrf_token %}
      <div class="container">
        <div class="row">
          <div class="col-md-3">
            {{ form.nombre.label_tag }}
          </div>
          <div class="col-md-9">
            {{ form.nombre }}
          </div>
        </div>
        <div class="row">
          <div class="col-md-3">
          </div>
          <div clases="col-md-9">
            {{ form.nombre.errors|linebreaks }}
          </div>
        </div>
        <div class="row">
          <div class="col-md-3">
            {{ form.descripcion.label_tag }}
          </div>
          <div class="col-md-9">
            {{ form.descripcion }}
          </div>
        </div>
        <div class="row">
          <div class="col-md-3">
          </div>
          <div class="col-md-9">
            {{ form.descripcion.errors|linebreaks }}
          </div>
        </div>
```

```
        <div class="row">
          <div class="col-md-3"></div>
          <div class="col-md-9">
            <input type="submit" value="Crear"/>
          </div>
         </div>
        </div>
      </form>
    </div>
</div>
{% endblock %}
```

Con lo que logramos lo siguiente:



Actualizar (Update)

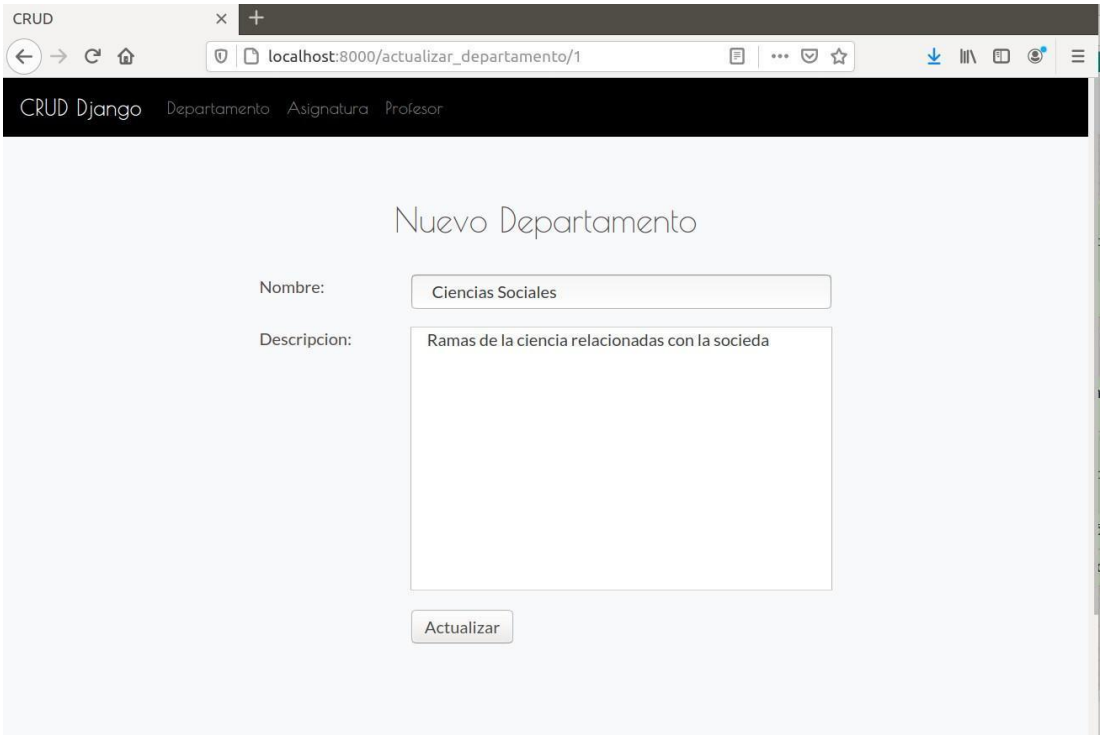
Cada botón **Editar** de nuestro template **comuna01/departamento.html** nos lleva a la siguiente página, que es muy similar al caso de `Crear`:

comuna01/departamento_form_actualizar.html

```
{% extends 'comuna01/base.html' %}

{% block main_content %}
<div class="container-fluid bg-light custom-section crud-form">
  <div clase="row justify-content-center align-items-center h-100">
    <h2 class="text-center">Nuevo Departamento</h2>
   </div><br>
   <div class="row justify-content-center align-items-center">
     <form method="post" action="" novalidate>
       {% csrf_token %}
     <div class="container">
       <div class="row">
         <div class="col-md-3">
           {{ form.nombre.label_tag }}
         </div>
         <div class="col-md-9">
           {{ form.nombre }}
         </div>
       </div>
       <div class="row">
         <div class="col-md-3">
         </div>
         <div class="col-md-9">
           {{ form.nombre.errors|linebreaks }}
         </div>
       </div>
       <div class="row">
         <div class="col-md-3">
           {{ form.descripcion.label_tag }}
         </div>
         <div class="col-md-9">
           {{ form.descripcion }}
         </div>
       </div>
       <div class="row">
         <div class="col-md-3">
         </div>
         <div class="col-md-9">
           {{ form.descripcion.errors|linebreaks }}
         </div>
       </div>
       <div class="row">
         <div class="col-md-3">
         </div>
         <div class="col-md-9">
           <input type="submit" value="Actualizar"/>
         </div>
       </div>
     </div>
   </form>
  </div>
</div>
{% endblock %}
```

Lo que resulta en:



Borrar (Delete)

En el caso de **Borrar** lo principal es que los botones del archivo **X** del template **comuna01/departamento.html** envíen al servidor la instrucción de borrar el registro correspondiente. Esto se logra con:

```
...
<a class="btn btn-danger btn-sm"
  href="{% url 'comuna01:url_borrar_departamento' pk=datum.id %}">X</a>
...
```

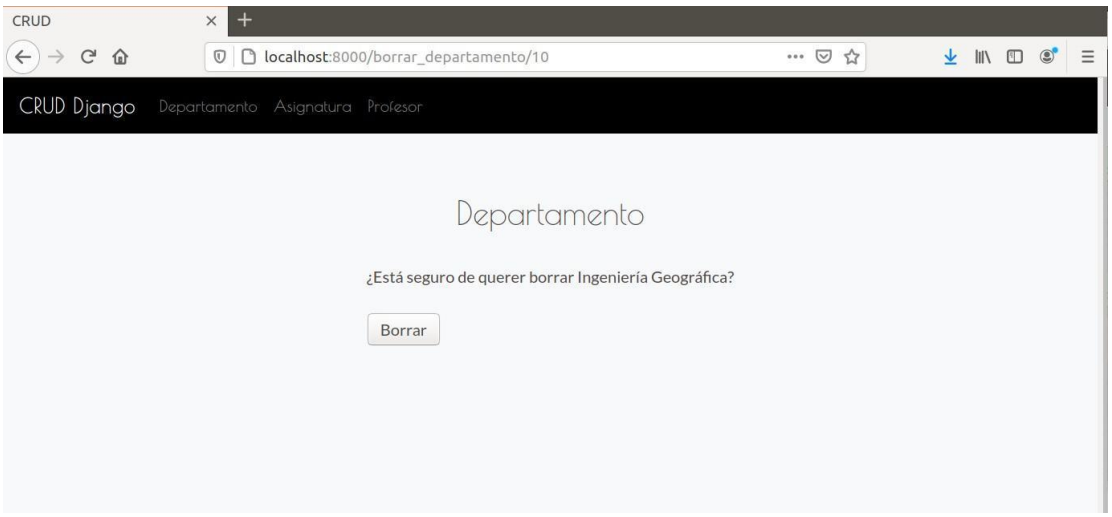
El código que está entre { } se entenderá mejor una vez mostremos los códigos para las vistas, más abajo.

En este caso incorporamos un template para confirmar que el usuario realmente desea borrar el registro en cuestión y evitar cometer errores.

```
{% extends 'comuna01/base.html' %}

{% block main_content %}
<div class="container-fluid bg-light custom-section crud-form">
  <div class="row justify-content-center align-items-center h-100">
    <h2 class="text-center">Departamento</h2>
  </div><br>
  <div class="row justify-content-center align-items-center">
    <form method="post"
          action="{% url 'comuna01:url_borrar_departamento' pk=datum.id
          %}" novalidate>
            {% csrf_token %}
            ¿Está seguro de querer borrar {{datum.nombre}}?<br><br>
      <input type="submit" name="" value="Borrar">
    </form>
  </div>
</div>
{% endblock %}
```

que tiene el siguiente aspecto:



VIEWS

En términos de vistas, utilizamos a continuación lo que se llama **Vistas Basadas en Funciones**, lo que corresponde a programar manualmente dentro de **comuna01/views.py** de la aplicación Django el procesamiento del request HTTP recibido por el servidor desde el cliente web. En apartados posteriores veremos otra manera de realizar estas mismas vistas con un acercamiento del tipo **Vistas Basadas en Clases** que permite utilizar clases que simplifican la implementación de una vista.

Para la vista **Leer**, que muestra el listado de todos los registros de nuestra tabla **departamento**, implementamos la función:

```
def lista_departamentos(request):
    template = 'comuna01/departamento.html'
    deptos = Departamento.objects.all().order_by('id')
    return render(request, template,
                  {'data': deptos})
```

Lo que esta función realiza es principalmente procesar el request obtenido y hacer una consulta de todos los objetos de la tabla **departamento**. Luego de esto la función **render()** los datos resultantes de la consulta.

Para la vista **Crear**, debemos implementar la función:

```
def crear_departamento(request):
    template = 'comuna01/departamento_form_crear.html'
    form = DepartamentoForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('comuna01:url_departamento')
    return render(request, template,
                  {"form": form})
```

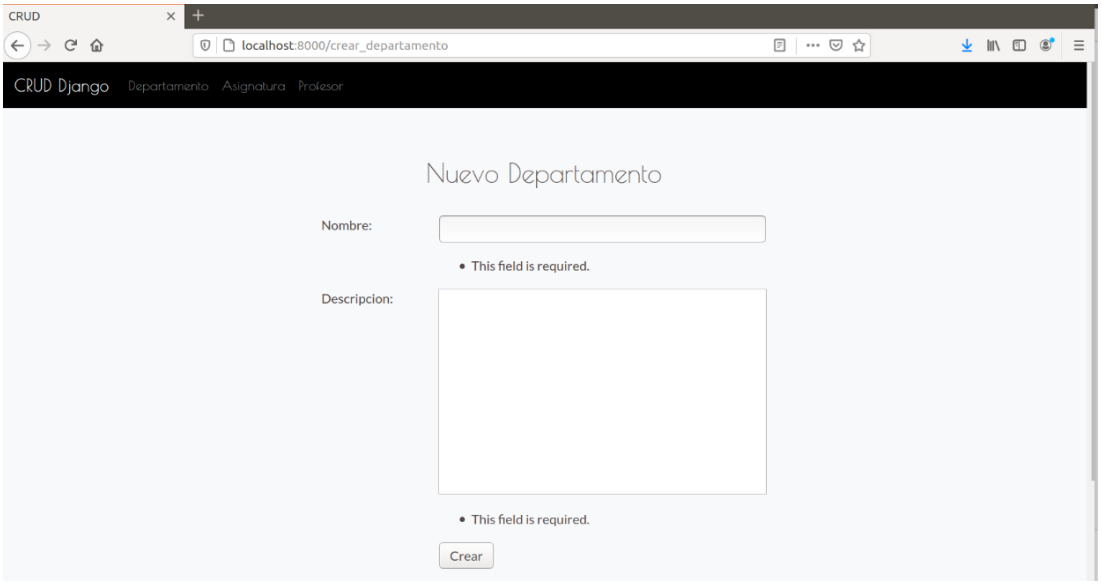
Que utiliza el formulario creado para este efecto en **comuna01/forms.py** el que se ve de la siguiente forma:

```
from django import forms
from .models import Departamento

class
    DepartamentoForm(forms.ModelForm):
    class Meta:
       model =
       Departamento
       widgets = {
         'nombre': forms.TextInput(attrs={'class': 'col-md-12'}),
         'descripcion': forms.Textarea(attrs={'class':
         'col-md-12'}),
       }
       fields = '_all_'
```

De esta manera la función **crear_departamento()** crea una instancia de la clase **DepartamentoForm** y lo envía al template **comuna01/departamento_form_crear.html.**

Django trabaja internamente, entre el **Model**, **Form** y **View** para validar y realizar la inserción de datos si los datos son válidos. En caso contrario, se mostrará el formulario con los mensajes de error correspondientes. Por ejemplo:



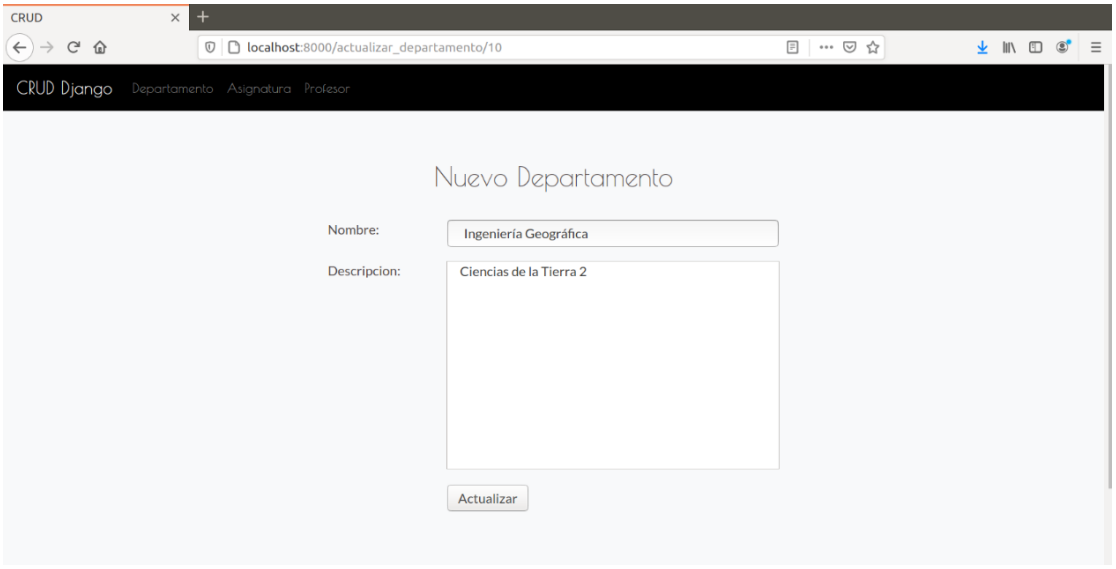
Similarmente, para la vista **Actualizar** se utiliza la siguiente función:

```
def actualizar_departamento(request, pk):
    template =
    'comuna01/departamento_form_actualizar.html' depto =
    get_object_or_404(Departamento, pk=pk)
    form = DepartamentoForm(request.POST or None,
    instance=depto) if form.is_valid():
        form.save()
         return
    redirect('comuna01:url_departamento') return
    render(request, template,
                   {"form": form})
```

En este caso utilizamos instanciamos la misma clase de formulario (**DepartamentoForm**), pasando como argumentos los datos del registro que solicitamos editar con el botón **Editar** en nuestra página del listado de departamentos:

```
...
<a class="btn btn-danger btn-sm"
href="{% url 'comuna01:url_actualizar_departamento' pk=datum.id %}">Editar</a>
...
```

De esta manera el formulario presenta los datos del registro que se deseó editar. Junto con esto nos permite actualizar información y realizar el request HTTP para su actualización por el servidor en la base de datos con la función **actualizar_departamento()**. Los datos son validados al igual que en el caso de la operación **Crear**.



Para el caso de la operación **Borrar**, utilizamos la función **borrar_departamento()**, que es muy similar a la función **actualizar_departamento()**, con la diferencia que la actualización realizada en este caso es borrar el registro con **depto.delete()**:

```
def borrar_departamento(request, pk):
    template = 'comuna01/departamento_form_borrar.html'
    depto = get_object_or_404(Departamento, pk=pk)
    if request.method == 'POST':
        depto.delete()
        return redirect('comuna01:url_departamento')
    return render(request, template,
                  {"datum": depto})
```

URLS

La definición de urls para lograr que todas las funciones de nuestras operaciones CRUD operen correctamente las realizamos en el archivo **comuna01/urls.py**, que queda como se muestra a continuación:

```
from django.urls import path

  # importing views from views..py
from .views import lista_departamentos
from .views import crear_departamento
from .views import
actualizar_departamento from .views
import borrar_departamento from .views
import asignatura
from .views import profesor
from .views import index
app_name = 'comuna01'

urlpatterns = [

       path('', index, name="url_comuna01"),
```

```
        path('departamento', lista_departamentos,
        name="url_departamento"),
        path('crear_departamento', crear_departamento,
        name="url_crear_departamento"),
        path('actualizar_departamento/<int:pk>', actualizar_departamento,
        name="url_actualizar_departamento"),
        path('borrar_departamento/<int:pk>', borrar_departamento,
        name="url_borrar_departamento"),
        path('asignatura', asignatura, name="url_asignatura"),
        path('profesor', profesor, name="url_profesor"),
]
```

Nótese que en el caso de las rutas para **Actualizar** y **Borrar** se pasa el parámetro de tipo número entero en la ruta con el identificados del elemento a ser editado o borrado, respectivamente. Este indentificador (**pk**) es pasado por el framework como un argumento a las funciones respectivas: **actualizar_departamento(request, pk), borrar_departamento(request, pk)**, además del argumento que lleva toda la información del request http.

El atributo **name** en cada una de las rutas es debido a que estamos utilizando **namespace** en el archivo **urls.py** del proyecto base para dividir nuestras distintas aplicaciones en el caso que existan más de ellas. Así también podemos utilizar la sintaxis simplificada que mostramos más arriba (**comuna01:url_departamento**) para referenciar nuestros templates en rutas determinadas de la aplicación. el archivo **urls.py** del proyecto principal, con este propósito, queda de la siguiente forma:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('comuna01.urls', namespace='comuna01'))
]
```

Referencias

[1] Django Documentation https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.

[4] Django CRUD Tutorial – Operations and Application Development https://data-flair.training/blogs/django-crud-example/