TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- CRUD en Django
- Creación de una aplicación Django para CRUD
- Interacción entre aplicaciones, modelos y vistas
- Manejo de token de seguridad CSRF

CRUD EN DJANGO

En este apartado se profundizará sobre la versatilidad de Django al permitir desarrollar aplicaciones con operaciones CRUD (Create, Read, Update and Delete), llevadas a cabo en una base de datos.

CREACIÓN DE UNA APLICACIÓN DJANGO PARA CRUD

Para crear un nuevo proyecto, utilizamos la siguiente instrucción:

```
1 django-admin startproject linkdump
```

La aplicación se va a llamar `linkdump`, el comando anterior crea un directorio `linkdump` en el que podemos encontrar los siguientes ficheros:

- `__init__.py:` Define nuestro directorio como un módulo Python válido.
- `manage.py:` Utilidad para gestionar nuestro proyecto, como arrancar servidor de pruebas, sincronizar modelos, entre otros.
- `settings.py:` Configuración del proyecto.
- `urls.py:` Gestión de las URLs. Este fichero sería el controlador de la aplicación, mapea las URL entrantes a funciones Python definidas en módulos.

A continuación se van a definir los parámetros de acceso a la base de datos en el gestor de base de datos mysql. Para ello se modifican algunas líneas del fichero `settings.py`:

```
DATABASES = {
'default': {
    'ENGINE': 'mysql',
    'NAME': 'bdurl',
    'USER': 'bduser',
    'PASSWORD': 'passuserbd',
    'HOST': '',
    'PORT': '',
}
```

Suponemos también que la base de datos ya ha sido creada. Y creamos las tablas necesarias para la administración de nuestra página con el siguiente comando:

```
python manage.py syncdb
```

Vamos a realizar una aplicación simple que nos permita gestionar una tabla con información de enlaces URL. Se llamará **`linktracker`**:

```
python manage.py startapp linktracker
```

Crearemos el modelo de datos de nuestra aplicación, que consistirá en una tabla donde guardaremos dos campos: **`link_description`** y **`link_url`**. Para ello añadimos en el fichero **`linktracker/models.py:`**

```
from django.db import models
# Create your models here.
class Link (models.Model):
    link_description = models.CharField(max_length=200)
    link_url = models.CharField(max_length=200)
```

Añadimos nuestra aplicación (**`linkdump.linktracker`**) en la lista **`INSTALLED_APPS`**, que encontramos en el fichero `settings.py`, y volvemos a actualizar nuestra base de datos para crear la nueva tabla:

```
python manage.py syncdb
```

ACTIVANDO EL SITIO DE ADMINISTRACIÓN

La aplicación de administración de nuestro sitio no está activada por defecto; para hacerlo tenemos que descomentar de la lista `INSTALLED_APPS`, que encontramos en `settings.py` la línea:

```
'django.contrib.admin'
```

Y volvemos a actualizar nuestra base de datos para crear los elementos necesarios para la aplicación de administración:

```
python manage.py syncdb
```

Para que funcione nuestra página de administración tenemos que descomentar en el fichero urls.py la línea referida a la aplicación admin:

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()
urlpatterns = patterns('',
    url (r'^admin/', include(admin.site.urls)),
)
```

Activamos nuestro servidor web, y probamos la página de administración:

```
python manage.py runserver
```

El siguiente paso es hacer que nuestra tabla con la información de los links pueda ser gestionada desde la página de administración. Para ello creamos un nuevo fichero llamado `admin.py` dentro de la carpeta de nuestra aplicación `linktracker`, con el siguiente contenido:

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

AÑADIENDO FUNCIONES A NUESTRA PÁGINA

Tenemos que ir añadiendo a nuestro controlador las acciones que se realizan al acceder a determinadas URL. Para ello, vamos a crear nuestra primera vista, donde mostraremos los datos de nuestra tabla. Modificamos el fichero urls.py, y añadimos lo siguiente:

```
1 url (r'^links/$', 'linkdump.linktracker.views.list'),
```

Gran parte de las vistas que vamos a crear utilizan una plantilla html (template). Para guardarlas crea un directorio llamado template dentro de tu aplicación, y añádelo a la lista `TEMPLATE_DIRS` que encontrarás en el fichero `settings.py`. Para crear la vista list añadimos el siguiente contenido al fichero `linktracker/views.py`:

```
 1 from django.core.context_processors import csrf
 2 from linkdump.linktracker.models import Link
 3 from django.template import Context, loader
 4 from django.shortcuts import render_to_response
 5 def list(request):
 6     link_list = Link.objects.all()
 7     return render_to_response(
 8         'links/list.html',
 9         {'link_list': link_list}
10     )
```

Y creamos la plantilla en `linktracker/template/links/list.html`:

```
 1 {% if link_list %}
 2     <ul>
 3          {% for link in link_list %}
 4              <li><a href='{{ link.link_url }}'>
 5                  {{link.link_description}}</a></li>
 6          {% endfor %}
 7     </ul>
 8 {% else %}
 9     <p>No links found.</p>
10 {% endif %}
```

Inicia el servidor web y accede a http://127.0.0.1/:8000/links, donde se verán los resultados.

PREPARANDO LA APLICACIÓN CRUD (CREATE, READ, UPDATE Y DELETE)

Hasta ahora se tiene una vista que nos muestra los links que tenemos guardados en nuestra tabla. A continuación necesitamos modificarla para que acepte parámetros get, de tal forma que podamos indicarles qué operación queremos realizar: añadir, modificar, borrar. Para ello, en `linktracker/views.py` tenemos que realizar dos cambios: cambiamos la definición del método `list`, y añadimos el parámetro message para que se transmita en cada llamada. Quedaría así:

```
from linkdump.linktracker.models import Link
from django.template import Context, loader
from django.shortcuts import render_to_response
def list(request, message = ""):
    link_list = Link.objects.all()
    return render_to_response(
        'links/list.html',
        {'link_list': link_list,'message': message}
    )
```

Y modificamos la plantilla:

```
{% if message %}
 <b>{{ message }}</b>
 <p>
 {% endif %}
 {% if link_list %}
     <table>
     {% for link in link_list %}
         <tr bgcolor='{% cycle FFFFFF,EEEEEE as rowcolor %}'>
             <td><a href='{{ link.link_url }}'>{{ link.link_description
}}</a></td>
             <td><a href='/links/edit/{{ link.id }}'>Edit</a></td>
             <td><a href='/links/delete/{{ link.id }}'>Delete</a></td>
         </tr>
     {% endfor %}
     </table>
     <p>
 {% else %}
     <p>No links found.</p>
 {% endif %}
 <p>
 <a href='/links/new'>Add Link</a>
```

AÑADIR ENLACES

Para dar la funcionalidad de añadir y guardar un nuevo registro en la base de datos. Añadimos dos nuevas líneas en el fichero `urls.py`:

```
url (r'^links/new', 'linkdump.linktracker.views.new'),
url (r'^links/add', 'linkdump.linktracker.views.add'),
```

Y definimos los dos nuevos métodos en nuestra vista (`linktracker/views.py`):

```
def new(request):
    return render_to_response(
        'links/form.html',
        {'action': 'add',
        'button': 'Add'}
    )
def add(request):
    link_description = request.POST["link_description"]
    link_url = request.POST["link_url"]
    link = Link(
        link_description = link_description,
        link_url = link_url
    )
    link.save()
    return list(request, message="Link added!")
```

Y creamos la plantilla `linktrcker/template/links/form.html`:

```
<form action="/links/{{ action }}/" method="post">
    Description:
    <input name=link_description value="{{ description }}"><br />
    URL:
    <input name=link_url value="{{ url }}"><br />
    <input type=submit value="{{ button }}">
</form>
```

MODIFICANDO ENLACES

Para modificar la información de un enlace, en este caso en la URL, tenemos que indicar qué enlace vamos a modificar, y añadimos una nueva acción en `urls.py`:

```
url(r'^links/edit/(?P<id>\d+)','linkdump.linktracker.views.edit'),
```

Creamos un nuevo método en nuestra vista (`linktracker/views.py`). Toma en cuenta que vamos a usar la misma plantilla que en el punto anterior:

```
def edit(request, id):
    link = Link.objects.get(id=id)
    return render_to_response(
        'links/form.html',
        {'action': 'update/' + id,
        'button': 'Update',
        'description': link.link_description,
        'url': link.link_url
        }
    )
```

Por último, hacemos algo similar para la acción update que modifica en la base de datos el dato, añadimos en `urls.py`:

```
url(r'^links/update/(?P<id>\d+)','linkdump.linktracker.views.update'),
```

Y en la vista (`linktracker/views.py`) un nuevo método:

```
def update(request, id):
    link = Link.objects.get(id=id)
    link.link_description = request.POST["link_description"]
    link.link_url = request.POST["link_url"]
    link.save()
    return list(request, message="Link updated!")
```

BORRANDO ENLACES

Para la opción de borrar un enlace, volvemos a añadir una nueva acción al fichero `urls.py`:

```
url (r'^links/delete/(?P<id>\d+)', 'linkdump.linktracker.views.delete'),
```

Y añadimos el método correspondiente en `linktracker/views.py`:

```
def delete(request, id):
    Link.objects.get(id=id).delete()
    return list(request, message="Link deleted!")
```

INTERACCIÓN ENTRE APLICACIONES, MODELOS Y VISTAS

Las aplicaciones web de Django acceden y administran los datos a través de objetos de Python a los que se hace referencia como modelos. Los modelos definen la estructura de los datos almacenados, incluidos

los tipos de campo y los atributos de cada campo, como su tamaño máximo, valores predeterminados, lista de selección de opciones, texto de ayuda para la documentación, texto de etiqueta para formularios, entre otros. La definición del modelo es independiente de la base de datos subyacente, se puede elegir una de entre varias como parte de la configuración de su proyecto. Una vez que haya elegido la base de datos que desea usar, no necesita hablar directamente con ella. Simplemente escriba la estructura de su modelo y el código, y Django se encargará de todo el trabajo para comunicarse con la base de datos.

En un sitio web tradicional basado en datos, una aplicación web espera peticiones HTTP del explorador web (o de otro cliente). Cuando se recibe una petición, la aplicación elabora lo que se necesita basándose en la URL, y posiblemente en la información incluida en los datos `POST` o `GET`. Dependiendo de lo que se requiera, quizás pueda entonces leer o escribir información desde una base de datos, o realizar otras tareas para satisfacer la petición. La aplicación devolverá a continuación una respuesta al explorador web, con frecuencia creando dinámicamente una página HTML para que el explorador la presente insertando los datos recuperados en marcadores de posición dentro de una plantilla HTML.

Las aplicaciones web de Django normalmente agrupan el código que gestiona cada uno de estos pasos en ficheros separados:

URLs: aunque es posible procesar peticiones de cada URL individual vía una función individual, es mucho más sostenible escribir una función de visualización separada para cada recurso. Se usa un mapeador URL para redirigir las peticiones HTTP a la vista apropiada basándose en la URL de la petición, y éste también puede emparejar patrones de cadenas o dígitos específicos que aparecen en una URL, y pasarlos a la función de visualización como datos.

Vista (View): es una función de gestión de peticiones que recibe peticiones HTTP, y devuelve respuestas HTTP. Acceden a los datos que necesitan para satisfacer las peticiones por medio de modelos, y delegan el formateo de la respuesta a las plantillas ("templates").

Modelos (Models): son objetos de Python que definen la estructura de los datos de una aplicación, y proporcionan mecanismos para gestionar (añadir, modificar y borrar) y consultar registros en la base de datos.

Plantillas (Templates): es un fichero de texto que define la estructura o diagrama de otro fichero (tal como una página HTML), con marcadores de posición que se utilizan para representar el contenido real. Una vista puede crear dinámicamente una página usando una plantilla, rellenándola con datos de un modelo. Una plantilla se puede usar para definir la estructura de cualquier tipo de fichero; no tiene por qué ser HTML.

Django se refiere a este tipo de organización como arquitectura Modelo Vista Plantilla "Model View Template (MVT)". Tiene muchas similitudes con la arquitectura Modelo Vista Controlador (Model View Controller).

MANEJO DE TOKEN DE SEGURIDAD CSRF

El paquete django.contrib.csrf provee protección contra Cross-site request forgery (CSRF) (falsificación de peticiones inter-sitio).

CSRF, también conocido como "session riding" (montado de sesiones), es un exploit de seguridad en sitios web. Se presenta cuando un sitio web malicioso induce a un usuario a cargar sin saberlo una URL desde un sitio al cual dicho. Veamos un ejemplo simple de CSRF:

Supongamos que posees una cuenta de webmail en example.com. Este sitio proveedor de webmail tiene un botón `Log Out`, el cual apunta a la URL `example.com/logout` — esto es, la única acción que necesitas realizar para desconectarte (log out) es visitar la página example.com/logout.

Un sitio malicioso puede obligarte a visitar la URL example.com/logout, incluyendo esa URL como un <iframe> oculto en su propia página maliciosa. De manera que si estás conectado (logged in) a tu cuenta de webmail del sitio example.com, y visitas la página maliciosa, te desconectará de example.com.

Este tipo de exploit puede sucederle a cualquier sitio que confía en sus usuarios, tales como un sitio de un banco, o uno de comercio electrónico.

UN EJEMPLO MÁS COMPLEJO DE CSRF

En el ejemplo anterior, el sitio example.com tenía parte de la culpa debido a que permitía que se pudiera solicitar un cambio de estado (la desconexión del sitio) mediante el método HTTP `GET`. Es una mejor práctica requerir el uso de un `POST` HTTP para cada petición que cambie el estado en el servidor. Pero aun los sitios web que requieren el uso de `POST` para acciones que signifiquen cambios de estado son vulnerables a CSRF.

Supongamos que example.com ha mejorado su funcionalidad de desconexión, de manera que "Log Out" es ahora un botón de un `<form>` que es enviado vía un `POST` a la URL `example.com/logout`. Adicionalmente, el `<form>` de desconexión incluye un campo oculto:

```
<input type="hidden" name="confirm" value="true" />
```

Esto asegura que un simple `POST` a la URL example.com/logout no desconectará a un usuario; para que los usuarios puedan desconectarse, deberán enviar una petición a `example.com/logout` usando `POST`, y enviar la variable `POST` confirm con el valor `'true'`. Bueno, aun con dichas medidas extra de seguridad, este esquema también puede ser atacado mediante CSRF, pues la página maliciosa sólo necesita hacer un poco más de trabajo. Los atacantes pueden crear un formulario completo que envíe su petición a tu sitio, ocultar el mismo en un `<iframe>` invisible, y luego usar JavaScript para enviar dicho formulario en forma automática.

Entonces, ¿cómo puede tu sitio defenderse de este exploit? El primer paso es asegurarse que todas las peticiones `GET` no poseen efectos colaterales. De esa forma, si un sitio malicioso incluye una de tus páginas como un `<iframe>`, esto no tendrá un efecto negativo.

Esto nos deja con las peticiones `POST`. El segundo paso es dotar a cada `<form>` que se enviará vía `POST` un campo oculto cuyo valor sea secreto, y sea generado en base al identificador de sesión del usuario. Entonces, cuando se esté realizando el procesamiento del formulario en el servidor, se comprobará dicho campo secreto, y generará un error si dicha comprobación no es exitosa.

Esto es precisamente lo que hace la capa de prevención de CSRF de Django, tal como se explica a continuación:

El paquete django.contrib.csrf contiene sólo un módulo: `middleware.py`. Este, a su vez, contiene una clase middleware Django: `CsrfMiddleware` la cual implementa la protección contra CSRF.

Para activar esta protección, agrega

```
1 'django.contrib.csrf.middleware.CsrfMiddleware'
```

a la variable de configuración `MIDDLEWARE_CLASSES` en el archivo de configuración. Este middleware necesita procesar la respuesta después de `SessionMiddleware`, así que `CsrfMiddleware` debe aparecer antes que `SessionMiddleware` en la lista (esto es debido que el middleware de respuesta es procesado de atrás hacia adelante). Por otra parte, debe procesar la respuesta antes que la misma sea comprimida o alterada de alguna otra forma, de manera que `CsrfMiddleware` debe aparecer después de `GZipMiddleware`. Una vez que has agregado eso a tu `MIDDLEWARE_CLASSES`, ya estás listo.

`CsrfMiddleware` trabaja de la siguiente manera, realizando estas dos cosas:

- Modifica las respuestas salientes a peticiones, agregando un campo de formulario oculto a todos los formularios `POST`, con el nombre csrfmiddlewaretoken y un valor que es un hash del identificador de sesión más una clave secreta. El middleware no modifica la respuesta si no existe un identificador de sesión, de manera que el costo en rendimiento es despreciable para peticiones que no usan sesiones.
- Para todas las peticiones `POST` que porten la cookie de sesión, comprueba que csrfmiddlewaretoken esté presente y tenga un valor correcto. Si no cumple estas condiciones, el usuario recibirá un error HTTP 403. El contenido de la página de error es el mensaje "Cross Site Request Forgery detected. Request aborted."

Esto asegura que solamente se puedan usar formularios que se hayan originado en tu sitio web para enviar datos vía `POST` al mismo. Este middleware deliberadamente trabaja solamente sobre peticiones HTTP `POST` (y sus correspondientes formularios `POST`). Las peticiones `GET` no deberían tener efectos colaterales; es trabajo del programador asegurarlo.

Las peticiones `POST` que no estén acompañadas de una cookie de sesión no son protegidas, simplemente porque no tiene sentido hacerlo, pues un sitio web malicioso podría de todas formas generar ese tipo de peticiones.

Para evitar alterar peticiones no HTML, el middleware revisa la cabecera `Content-Type` de la respuesta antes de modificarla. Sólo modifica las páginas que son servidas como `text/html`, o `application/xml+xhtml`.

Para aprovechar la protección del CSRF, siga estos pasos:

1. El middleware CSRF se activa por defecto en la configuración de MIDDLEWARE. Si anula esa configuración, recuerde que `'django.middleware.csrf.CsrfViewMiddleware'` debe aparecer antes que cualquier middleware de vista que suponga que se han abordado los ataques CSRF.
2. Si lo desactivó, lo que no es recomendable, puede usar `csrf_protect()` en vistas particulares que desea proteger.

3. En cualquier plantilla que use un formulario `POST`, use la etiqueta `csrf_token` dentro del elemento `<form>` si el formulario es para una URL interna. Por ejemplo:

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

Esto no debe hacerse para los formularios `POST` que apuntan a URLs externas, ya que causaría que el token del CSRF se filtre, lo que provocaría una vulnerabilidad.

4. En las funciones de vista correspondientes, asegúrese de que `RequestContext` se use para representar la respuesta, de modo que `{% csrf_token %}` funcione correctamente. Si está utilizando la función `render()`, vistas genéricas, o aplicaciones `contrib`, ya está cubierto pues todos usan `RequestContext`.