## TEXT CLASS REVIEW

## TEMAS A TRATAR EN EL CUE

- Modelo Login/logout.
- Modelo Autorización.
- Modelo Auth de Django.
- ¿Cómo maneja la seguridad Django?
- ¿Cómo utilizar el modelo Auth de Django?
- Ejecutando las migraciones.
- Enrutamiento login/logout.
- Configuración en settings.py.
- LOGIN_REDIRECT_URL.
- LOGOUT_REDIRECT_URL.

### AUTENTICACIÓN Y AUTORIZACIÓN

### MODELO LOGIN/LOGOUT

Permite a los usuarios iniciar sesión (login) en un sistema que resuelve dos problemas: autenticación y autorización. La autenticación es el acto de verificar la identidad de un usuario, confirmando que es quien dice ser. Django integra una funcionalidad para inicio/cierre de sesión a través del sistema de autenticación de usuario que veremos más adelante.

### MODELO AUTORIZACIÓN

Luego de que un usuario se autentifique en un sistema, ocurre el proceso de autorización (verificación de permisos), el cual determina qué acciones tiene permitido realizar el usuario autenticado. Django incluye modelos para Users y Groups (una forma genérica de aplicar permisos a más de un usuario a la vez), permisos/indicadores (permissions/flags) que designan si un usuario puede realizar una tarea, formularios y vistas para iniciar sesión en los usuarios, y view tools para restringir el contenido. Los conceptos de autenticación y autorización de usuarios van de la mano, si un sistema está restringido a través del

inicio/cierre de sesión, los usuarios tienen que autenticarse antes de que ellos puedan ser autorizados para hacer uso del sistema.

**MODELO AUTH DE DJANGO**

El sistema de autenticación de Django incluye tanto autenticación como autorización. Maneja cuentas de usuario, grupos, permisos y sesiones de usuario basadas en cookies, sistema de `hashing` para contraseñas, formularios para acceso de usuarios o restricción de contenido, y un sistema de respaldo.

**CÓMO MANEJA LA SEGURIDAD DJANGO**

Django ofrece varias características relacionadas con la seguridad:

- **Protección de la escritura de sitios cruzados (XSS)**: los ataques XSS permiten a un usuario inyectar secuencias de comandos del lado del cliente en los navegadores de otros usuarios. Pueden originarse en cualquier fuente de datos no confiable, como cookies o servicios web. El uso de plantillas Django protege contra la mayoría de los ataques XSS. Sin embargo, es importante conocer qué protecciones proporciona y sus limitaciones.

- **Protección contra la falsificación de solicitudes en sitios cruzados (CSRF)**: los ataques CSRF permiten que un usuario malintencionado ejecute acciones utilizando las credenciales de otro usuario sin su conocimiento o consentimiento. Django tiene protección incorporada contra la mayoría de los tipos de ataques CSRF, siempre que la opción sea habilitada y utilizada debidamente. Sin embargo, como con cualquier técnica de mitigación, existen limitaciones. Por ejemplo: es posible deshabilitar el módulo CSRF globalmente, o para vistas particulares. Solo debe hacer esto si sabe lo que está haciendo.

- **Protección de inyección SQL**: la inyección SQL es un tipo de ataque en el que un usuario malicioso es capaz de ejecutar código SQL arbitrario en una base de datos. Esto puede dar lugar a la eliminación de registros o a la fuga de datos. Los conjuntos de consultas de Django están protegidos de la inyección de SQL, ya que sus consultas se construyen utilizando la parametrización de éstas. El código SQL de una consulta en Django se define por separado de los

parámetros. Dado que los parámetros pueden ser proporcionados por el usuario y, por lo tanto, inseguros, por lo general el gestor de base de datos no gestiona la validación de éstos.

- **Protección contra el clickjacking**: el `clickjacking` es un tipo de ataque en el que un sitio malicioso envuelve a otro sitio en un marco. Este ataque puede hacer que un usuario desprevenido sea engañado para que realice acciones no deseadas en el sitio objetivo. Django contiene protección contra clics en forma de `X-Frame-Options` middleware, que en un navegador compatible puede evitar que un sitio se represente dentro de un marco.

- **SSL/HTTPS**: siempre es mejor para la seguridad implementar su sitio detrás de HTTPS. Sin esto, es posible que los usuarios de redes malintencionadas detecten las credenciales de autenticación o cualquier otra información transferida entre el cliente y el servidor y, en algunos casos, los atacantes de red activos alteren los datos que se envían en cualquier dirección. Luego de habilitar la protección que ofrece el HTTPS, hay algunos pasos adicionales que se requieren seguir para configurar las directivas adecuadamente de tal manera que la protección sea efectiva.

- **Validación de la cabecera del host:** Django usa el encabezado Host proporcionado por el cliente para construir URL en ciertos casos. Si bien estos valores se desinfectan para evitar ataques de Cross Site Scripting, se puede usar un valor de Host falso para la falsificación de solicitudes entre sitios, ataques de envenenamiento de caché, y enlaces de envenenamiento en correos electrónicos. Debido a que incluso las configuraciones de servidor web aparentemente seguras son susceptibles a encabezados falsos de Host, Django valida los encabezados de Host con la configuración `ALLOWED_HOSTS` en el método: `django.http.HttpRequest.get_host()`.

- **Política de remisión**: los navegadores usan el encabezado `Referer` como una forma de enviar información a un sitio sobre cómo los usuarios llegaron allí. Al establecer una Política de referencia, puede ayudar a proteger la privacidad de sus usuarios, restringiendo en qué circunstancias se establece el encabezado `Referer`.

- **Política de apertura cruzada**: el encabezado de política de apertura de origen cruzado (COOP) permite a los navegadores aislar una ventana de nivel superior de otros documentos colocándolos en un grupo de contexto diferente para que no puedan interactuar directamente con la ventana de

nivel superior. Si un documento protegido por COOP abre una ventana emergente de origen cruzado, la propiedad `window.opener` de la ventana emergente será null.

**¿CÓMO UTILIZAMOS EL MODELO AUTH DE DJANGO?**

La configuración por defecto del sistema de autenticación de Django satisface las necesidades más comunes de los proyectos, maneja una gama razonablemente amplia de tareas, y tiene una implementación cuidadosa de contraseñas y permisos. La autenticación de Django proporciona autenticación y autorización a la vez y, en general, se les conoce como el sistema de autenticación, ya que estas funciones están estrechamente relacionadas.

- **El objeto User:** el núcleo del sistema de autenticación lo conforma los objetos de usuario, y representan a las personas que interactúan con el sistema. Se utilizan para habilitar funcionalidades como restringir el acceso, registrar perfiles de usuario, asociar contenido con creadores, entre otros. Solo existe una clase de usuario en el marco de autenticación de Django, es decir, 'superusers' o administradores y 'staff'. Los usuarios son solo objetos de usuario con atributos especiales establecidos, no diferentes clases de objetos de usuario. Los atributos básicos son: username, password, email, primer nombre y primer apellido. La forma más directa de crear usuarios es hacer uso de la función auxiliar `create_user()`, incluida en el sistema de autenticación, de la siguiente manera:

```
1 from django.contrib.auth.models import User
2 user = User.objects.create_user(juan,
3 juan@bubbles.com','juanpassword')
```

Sin embargo, si se ha instalado el admin de Django puede crear los usuarios interactivamente. Por otra parte, los superusuarios se pueden crear haciendo uso del comando `createsuperuser`, así:

```
1 $ python manage.py createsuperuser --username=jack --
2 email=jack@bubbles.com
```

Seguidamente, luego de ejecutar el comando, se le pedirá agregar la contraseña del superusuario.

- **Autenticando usuarios:** para verificar las credenciales de un usuario en específico el sistema de autenticación hace uso de `authenticate()`, tomando como parámetros el nombre del usuario (username), y la contraseña (password) para el caso predeterminado; las compara con cada backend de autenticación, y devuelve un objeto User si son válidas para un backend. Si las credenciales no son válidas para ningún backend, o si un backend genera Permiso denegado, devuelve `None`. Por ejemplo, la siguiente es una forma de bajo nivel para autenticar un conjunto de credenciales; lo utiliza `RemoteUserMiddleware`

```
from django.contrib.auth import authenticate
    user = authenticate(username=juan, password='secret')
   if user is not None:
      # credenciales autenticadas
   else:
     #  credenciales no autenticadas
```

- **Autenticación en solicitudes web:** Django hace uso de sesiones y middleware para conectar el sistema de autenticación a los objetos de solicitud. Estos proporcionan un atributo `request.user` en cada solicitud que representa al usuario actual. Si el usuario actual no ha iniciado sesión, este atributo se establecerá en una instancia de `AnonymousUser`; de lo contrario, será una instancia de `Usuario`. Por ejemplo:

```
if request.user.is_authenticated:
# opciones para Usuario autenticado
...
else:
# opciones para usuario anonimo
...
```

- **Inicio de sesión para un usuario:** Al un usuario ser autenticado se debe atar a la sesión actual, esto se logra a través de la función `login()`:

```
login(request, user, backend=None)
```

Esta función es utilizada para iniciar sesión de un usuario desde una vista, toma un objeto `HttpRequest` y un objeto User, la función guarda el ID del usuario en la sesión haciendo uso de la

funcionalidad session de Django. El siguiente ejemplo muestra cómo hacer uso de ambas funciones `authenticate()` y `login()`:

```
from django.contrib.auth import authenticate, login
def my_view(request):
username = request.POST['username']
password = request.POST['password']
user = authenticate(request, username=username, password=password)
if user is not None:
     login(request, user)
       # Redirigir a la página permitida
        ...
else:
      # Devolver un mensaje de error por acceso inválido
        ...
```

- Cierre de sesión para un usuario: para cerrar la sesión de un usuario que la ha iniciado vía `django.contrib.auth.login()`, se utiliza la función `django.contrib.auth.logout()` dentro de la vista. La función toma un objeto `HttpRequest`, y no devuelve ningún valor. Ejemplo:

```
from django.contrib.auth import authenticate, login
def my_view(request):
username = request.POST['username']
password = request.POST['password']
from django.contrib.auth import logout
def logout_view(request):
logout(request)
  # Redirigir con mensaje satisfactorio
```

Cuando se llama la función `logout()`, los datos de la sesión para la solicitud actual se limpian por completo. Esto es para evitar que otra persona use el mismo navegador web para iniciar sesión, y tener acceso a los datos de la sesión del usuario anterior.

**EJECUTANDO LAS MIGRACIONES**

Las migraciones son la forma en que Django propaga los cambios que realiza en sus modelos (agregar un campo, eliminar un modelo, entre otros) en el esquema de su base de datos. Están diseñados para ser en su mayoría automáticos, pero necesitará saber cuándo realizar migraciones, cuándo ejecutarlas, y los problemas comunes que podría encontrar.

Existen varios comandos que se utilizan para interactuar con las migraciones y el manejo del esquema de base de datos de Django:

- `migrate`: aplicar y desaplicar migraciones.
- `makemigrations`: crear nuevas migraciones en función de los cambios que haya realizado en sus modelos.
- `Sqlmigrate`: muestra las instrucciones SQL para una migración.
- `Showmigrations`: enumera las migraciones de un proyecto y su estado.

Las migraciones se pueden asociar a un sistema de control de versiones para el esquema de su base de datos. El comando `makemigrations` es responsable de empaquetar los cambios de un modelo en archivos de migración individuales, de forma análoga a las confirmaciones, y `migrate` es responsable de aplicarlos a su base de datos.

Los archivos de migración para cada aplicación se encuentran en el directorio de "migraciones" dentro de la aplicación, y están diseñados para comprometerse y distribuirse como parte de su código base. Django puede crear migraciones por sí solo, intente realizar algún cambio en los modelos de su app. Por ejemplo, puede agregar un campo y eliminar un modelo, ejecute seguidamente `makemigrations` así:

```
$ python manage.py makemigrations
```

```
Migrations for 'cars':
  cars/migrations/0005_auto.py:
    - Alter field owner on car
```

Al ejecutar el comando, los modelos se escanearán y compararán con las versiones contenidas actualmente en los archivos de migración, y luego se escribirá un nuevo conjunto de migraciones. Es importante leer el resultado para ver lo que `makemigrations` ha cambiado.

Al tener los nuevos archivos de migración, debe aplicarlos a su base de datos para asegurarse de que funcionen como se espera:

```
$ python manage.py migrate
```

```
Operations to perform:
  Apply all migrations: cars
Running migrations:
  Rendering model states... DONE
  Applying cars.0005_auto... OK
```

- Archivos de migración: son archivos con código Python, escritos en un estilo declarativo. Un archivo de migración básico luce de la siguiente manera:

```
from django.db import migrations, models
class Migration(migrations.Migration):
dependencies = [('migrations', '0001_initial')]
operations = [
        migrations.DeleteModel('Tribble'),
        migrations.AddField('Author', 'rating',
models.IntegerField(default=0)),
]
```

Al cargar un archivo de migración, Django busca la subclase `django.db.migrations.Migration` llamada Migración. Luego inspecciona este objeto en busca de cuatro atributos, de los cuales solo dos se usan la mayor parte del tiempo:

- dependencias, una lista de migraciones de las que depende.
- operaciones, una lista de clases de operación que definen lo que hace esta migración.

Las operaciones son un listado de instrucciones declarativas que le dicen a Django qué cambios de esquema deben realizarse. Django los escanea, y construye una representación en memoria de todos los cambios de esquema en todas las aplicaciones, y usa esto para generar el SQL que hace los cambios de esquema.

Esa estructura en memoria también se usa para determinar cuáles son las diferencias entre los modelos, y el estado actual de sus migraciones; Django ejecuta todos los cambios, en orden, en un conjunto de modelos en memoria para determinar el estado de sus modelos la última vez que

ejecutó `makemigrations`. Luego, usa estos modelos para compararlos con los de sus archivos `models.py` para determinar qué ha cambiado.

- **Migración inicial:** se refiere a las migraciones que crean la primera versión de las tablas de una aplicación. Estas se marcan con un atributo de clase `initial = True` en la clase de migración.
- **Revertir migraciones:** las migraciones se pueden revertir con el comando `migrate`, pasando el número de la migración anterior. Por ejemplo, para revertir la migración `cars.0001`:

```
$ python manage.py migrate cars 0001
```

```
Operations to perform:
  Target specific migration: 0001_auto, from cars
Running migrations:
  Rendering model states... DONE
  Unapplying cars.0005_auto... OK
```

- **Migración de datos:** además de cambiar el esquema de la base de datos, también puede usar migraciones para cambiar los datos en la propia base de datos, junto con el esquema. Las migraciones que modifican los datos suelen denominarse "migraciones de datos"; se escriben mejor como migraciones separadas, junto con las migraciones de su esquema. Django no puede generar automáticamente migraciones de datos, como lo hace con las migraciones de esquema. Los archivos de migración en Django se componen de Operaciones, y la operación principal que usa para las migraciones de datos es `RunPython`.

Para comenzar una migración de datos, se crea un archivo de migración vacío:

```
$ python manage.py makemigrations --empty nombre_de_app
```

El archivo lucirá de la siguiente manera:

```
from django.db import migrations
class Migration(migrations.Migration):
dependencies = [
     ('nombre_de_app', '0001_initial'),
]
operations = [
```

```
    ]
```

Posteriormente, todo lo que necesita hacer es crear una nueva función, y hacer que `RunPython` la use.

**ENRUTAMIENTO LOGIN/LOGOUT**

Django maneja una solicitud enrutando la ruta URL entrante a una función de vista. La función de visualización es responsable de devolver una respuesta al cliente que realiza la solicitud HTTP. Para dirigir la solicitud a una función de vista específica, Django analiza la configuración de su URL o `URLconf`. La plantilla de proyecto predeterminada define el `URLconf` en `urls.py`. El `URLConf` debe ser un módulo de Python que define un atributo llamado `urlpatterns`, que no es más que una lista de `django.conf.urls.url()`. Cada instancia de `url()` debe, como mínimo, definir una expresión regular para que coincida con la URL y un objetivo, que es una función de vista o un `URLconf` diferente. Si un patrón de URL apunta a una función de vista, es una buena idea darle un nombre para hacer referencia fácilmente al patrón más adelante. Un ejemplo de enrutamiento de inicio, y cierre de sesión, es el siguiente:

```
path('login/',
auth_views.LoginView.as_view(template_name='myapp/login.html'),
name='login')
path('logout/',
auth_views.LogoutView.as_view(template_name='myapp/login.html'),
name='logout')
```

Configuración en `settings.py`: la configuración requerida por el módulo de autenticación de usuarios de Django está incluida en el archivo `settings.py`. La configuración consiste en dos ítems que se ubican en la sección `INSTALLED_APPS`, y dos más en la sección `MIDDLEWARE`:

- `INSTALLED_APPS`:
  - `'django.contrib.auth'`: que contiene el núcleo del marco de autenticación, y sus modelos predeterminados.
  - `'django.contrib.contenttypes'`: que es el sistema de tipos de contenido de Django, que permite asociar permisos con los modelos que crea.
- `MIDDLEWARE`:

- `SessionMiddleware`: que gestiona sesiones a través de solicitudes.
- `AuthenticationMiddleware`: que asocia usuarios con solicitudes mediante sesiones. Con esta configuración en su lugar, ejecutar el comando `manage.py migrate` crea las tablas de base de datos necesarias para los modelos y permisos relacionados con la autenticación para cualquier modelo definido en sus aplicaciones instaladas.

**LOGIN_REDIRECT_URL**

Luego de que un usuario realiza un inicio de sesión satisfactorio, es necesario especificar a qué vista debe ser redirigido cuando la vista Login (`LoginView`) no especifica un parámetro `next` GET. Para este caso, se utiliza la configuración `LOGIN_REDIRECT_URL` a través de la cual se especifica la ruta. Al final del archivo `settings.py` se agrega la siguiente instrucción para redirigir al usuario a la vista principal. Por ejemplo:

```
1 LOGIN_REDIRECT_URL = “/”
```

El valor por defecto es: `'/accounts/profile/'`

**LOGOUT_REDIRECT_URL**

Esta directiva igualmente se especifica al final del archivo `settings.py`, y especifica el URL donde las solicitudes de `logout` deben ser redirigidas en caso de que la vista `LogoutView` no especifique el atributo `next_page`. Si el valor por defecto `None` está configurado, no ocurrirá ningún redireccionamiento y la vista `logout` será desplegada.