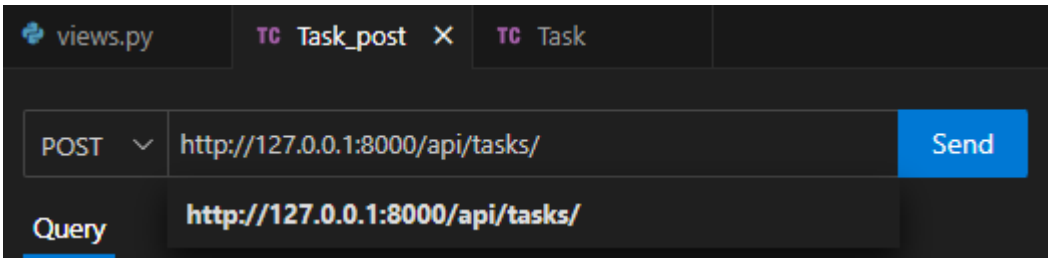
IMPLEMENTACIÓN DE POST

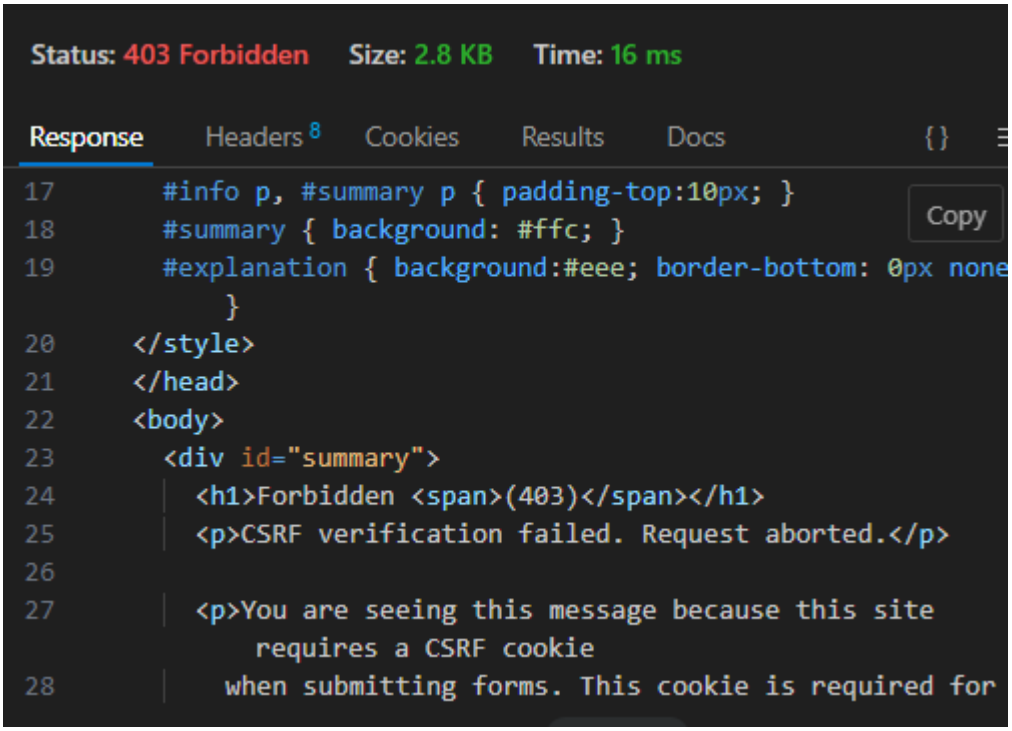
Consideraciones previas: Django rest framework, de forma intrínseca, añade cierta seguridad a las peticiones post debido a que estas se usan, como habíamos comentado anteriormente, para añadir nuevos datos en nuestra base de datos. Por tanto se debe tener especial cuidado con ello ya que puede ocurrir que se utilicen este tipo de peticiones para ingresar código malicioso. Veamos una prueba. Configuremos con un código muy básico la función “post” de nuestra vista. Por ejemplo de la siguiente manera:

```
def post (self, request):
    datos = {'message': 'Implementación de post'}
    return JsonResponse (datos)
```

E intentamos probar la petición en Thunder Client:



Si echamos un vistazo a la respuesta saldrá algo parecido:

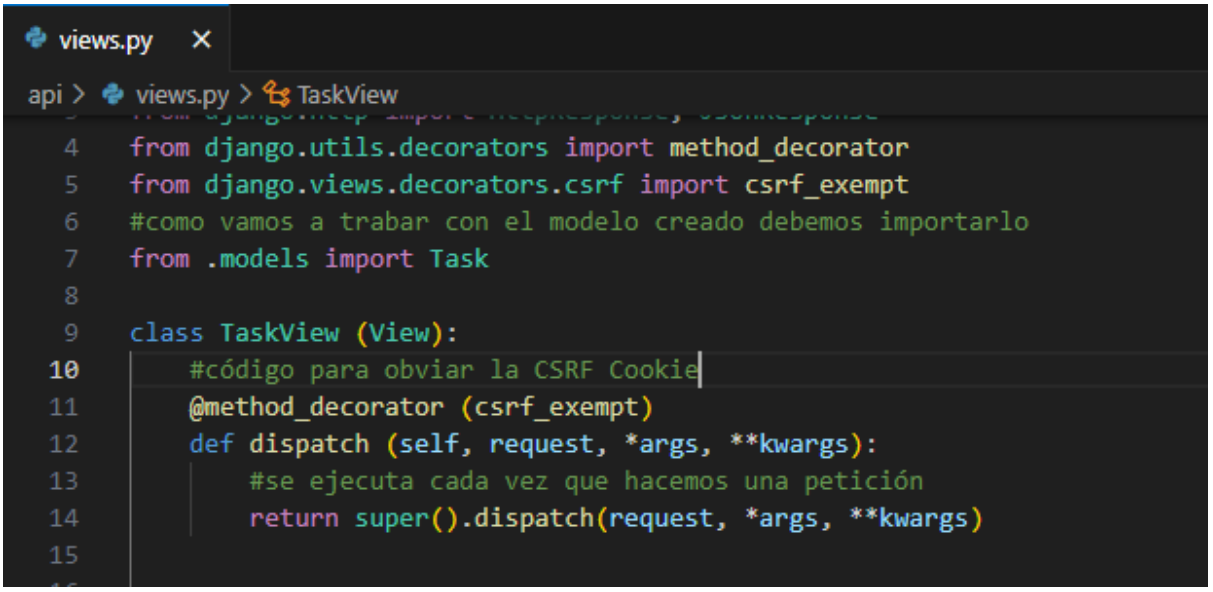


Hay un problema de acceso y hace referencia a **CSRF cookie**. El mensaje del terminal en concreto aparece: Forbidden (CSRF cookie not set.): /api/tasks/

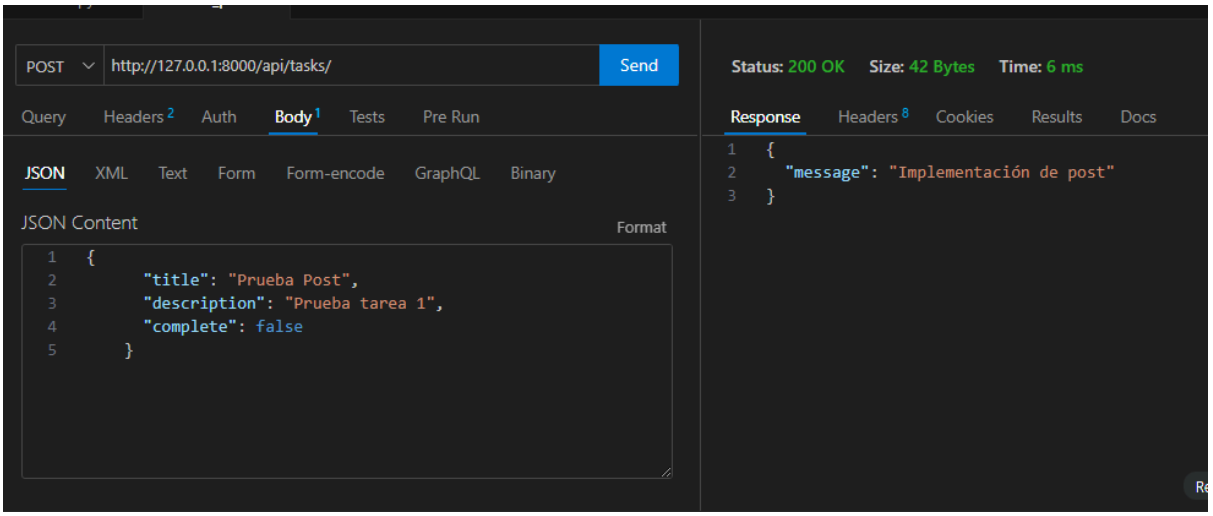
[Cross-Site Request Forgery (CSRF): qué es y cómo funciona (ciberseguridad.com)]

CSRF es una vulnerabilidad de seguridad web que permite a un atacante inducir a los usuarios a realizar acciones que no pretenden realizar. Los ataques de falsificación de solicitudes entre sitios (CSRF) son vulnerabilidades comunes de aplicaciones web que se aprovechan de la confianza que un sitio web ya ha otorgado a un usuario y su navegador.

Django, de forma predeterminada protege a las aplicaciones desarrolladas con un token de seguridad ante estas amenazas. Este token garantiza que la petición proviene de un sitio correcto y no de otro lugar. Como hasta el momento no se ha establecido ninguno, se rechaza la operación. No obstante, para continuar con la práctica (más adelante se establecerá dicho token), añadiremos un decorador y una pequeña función dentro de la vista para "saltarnos" esta protección que se ejecutará cada vez que se realice una petición POST. Obviamente en producción esto no lo haremos.



Si probamos ahora Thunder cliente enviando datos con el método POST veremos la respuesta que habíamos configurado.



Una vez comprobado y explicado esta eventualidad configuraremos la función post para insertar datos en la base de datos. Debes importar el módulo json para poder usar los métodos que incluye la función. Los print no son necesarios, se usan sólo por si quieres controlar qué datos se van cargando en cada momento.

```
def post (self, request):
    print (request.body)
    #cargamos el cuerpo del json donde recibimos los datos
    jd = json.loads(request.body)
    print (jd)
    #creamos un objeto en la base de datos, una nueva tarea a través del orm
    Task.objects.create (
        title = jd['title'],
        description = jd['description'],
        complete = jd['complete']
    )
    datos = {'message': 'Datos insertados'}
    return JsonResponse (datos)
```

Lanza nuevamente la petición POST a través del Thunder Cliente y observa la salida. Lanza también nuevamente la petición GET y comprueba que también se devuelve el nuevo registro.

IMPLEMENTACIÓN ACCESO A UNA SOLA TAREA

Modificaremos la vista para get que habíamos configurado.
Necesitamos una nueva ruta añadida en el fichero de rutas de nuestra api (api→urls.py)
El nuevo path a incluir es:

```
path ('tasks/<int:pk>', TaskView.as_view(), name = 'una_task')
```

Fíjate en el detalle <int:pk> Verás cuando modifiquemos la vista como interviene en la función. La lectura sería que esperamos como parámetro en la función get, un entero (que en este caso es la pk). pk es el nombre del parámetro que usaremos en la view. Es decir, si cambiáramos “pk” por “otra_cosa”, en la función get de la vista debemos usar “otra_cosa”

```python
def get (self, request,pk=0):
    #comprobamos si se ha pasado un id válido
    if pk > 0:
        #obtenemos la lista de tareas filtrada por ese ID
        tasks = list (Task.objects.filter (id=pk).values())
        if tasks:
            task = tasks[0]
            datos = {'message': 'OK!', 'tareas': task}
        else:
            datos = {'message':'No hay tareas con ese ID'}
    #si no se ha pasado ningún ID válido intenta listar todas las tareas
    else:

    #convertimos a listas para que los datos obtenidos se puedan serializar y convertir directamente a JSON.
    #podríamos haber escrito tasks = Task.objects.all(), pero esta fórmula nos generaría el error comentado es decir,
    #nos diría que los datos no son serializables y por tanto al intentar convertirlos a JSON daría error.
        tasks = list(Task.objects.values())
        if tasks:
            datos = {'message': 'OK!', 'tareas': tasks}
        else:
            datos = {'message': 'No hay tareas que hacer'}
    return JsonResponse(datos)
#los datos se devuelven en formato JSON
```

Observa el parámetro pk en la primera línea. Si en la ruta hubiéramos puesto por ejemplo:

```python
path ('tasks/<int:clave>', TaskView.as_view(), name = 'una_task')
```

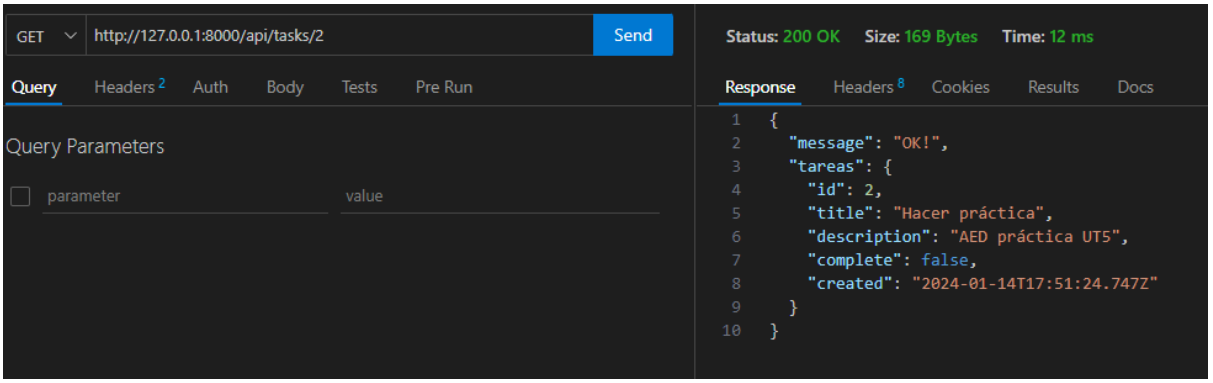
la línea quedaría:

```python
def get (self, request,clave=0):
… y cualquier referencia a pk debe cambiarse por "clave".
```

Insisto en esta cuestión para aclarar que el hecho de escribir <int:pk> para pasar como parámetro un id en la petición no significa que 'pk' sea una palabra reservada del framework sino simplemente el nombre del parámetro que se usará en la ruta.

Importante observar el uso y sintaxis de **Task.objects.filter (id=pk)** para obtener un solo valor filtrado.

Para probar la vista con Thunder Client, creamos una nueva request (no es necesario, pero así las tenemos todas diferenciadas y guardadas sin tener que estar cambiando), y hacemos la petición con la ruta que usamos para get, pero incluyendo un id específico:



IMPLEMENTACIÓN MÉTODO PUT

```python
def put (self, request, pk = 0):

    #Observa que el código para actualizar es una combinación del que hemos usado para POST
    #con el que hemos usado para recuperar una tarea específica.
    #Se cargan los nuevos datos y se llama al método save()

    jd = json.loads(request.body)
    tasks = list (Task.objects.filter (id=pk).values())
    if tasks:
        task = Task.objects.get(id=pk)
        task.title = jd['title']
        task.description = jd['description']
        task.complete = jd['complete']
        task.save()
        datos = {'message': 'Datos modificados'}
    else:
        datos = {'message': 'Tarea no encotrada'}
    return JsonResponse (datos)
```

Para probar el funcionamiento con Thunder usaremos la misma ruta que en las ocasiones anteriores, con parámetro como en el método get y con los datos a modificar en el body como en el método post.

IMPLEMENTACIÓN MÉTODO DELETE

```python
def delete (self, request, pk=0):
    tasks = list (Task.objects.filter (id=pk).values())
    if tasks:
        Task.objects.filter(id=pk).delete()
        datos = {'message': 'Datos borrados'}
    else:
        datos = {'message': 'Tarea no encotrada'}
    return JsonResponse (datos)
```

Prueba la función con Thunder Client usando la misma ruta con id y método DELETE

Actividad:

Añadir en el método get el código necesario para que al hacer una petición con el parámetro 0, se devuelva sólo la lista de tareas completadas.

Contesta: ¿Es necesario hacer alguna modificación en el fichero urls.py? En caso afirmativo, indica la ruta y en caso negativo explica por qué.