


Open   Opened 3 days ago by  **Rubén Montero**

# Un endpoint para crear usuarios

## Resumen

- Definiremos cómo es la petición REST mediante la que soportaremos el registro de nuevos usuarios
- Crearemos una función en `endpoints.py` que procese dicha petición
- Leerá del cuerpo del POST los atributos del nuevo usuario
- Codificará (*hashear*) la contraseña antes de almacenarla
- Guardará el nuevo usuario en la base de datos y enviará una respuesta HTTP de éxito si todo va bien

## Descripción

Nuestro API REST soportará la siguiente operación REST:

`/v1/users`

### POST

#### Resumen:

Crea un nuevo usuario

#### Parámetros

Nombre	En	Descripción	¿Obligatorio?	Tipo
email	body (json)	Correo electrónico	Sí	string
username	body (json)	Nombre de usuario	Sí	string
password	body (json)	Contraseña del usuario	Sí	string

#### Ejemplo de cuerpo de petición

```
{
  "email": "testemail@test.email.co.uk",
  "username": "Bill Doe",
  "password": "asdf1234"
}
```

#### Respuestas

Código	Descripción
201	El usuario se ha creado con éxito
400	La petición HTTP no se envió con todos los parámetros en el JSON, o alguno es de tipo incorrecto
409	Ya existe un usuario registrado con el <code>email</code> especificado

## Excelente

Sí.

¿Puedes redactar el comando `cURL` que envía una petición POST para registrar a un usuario (asumiendo que el servidor corre en *localhost*)

### ¡Pues claro!

```
curl -X POST http://localhost:8000/v1/users -d '{"email": "testemail@test.email.co.uk", "username": "Bill Doe", "pas
```

¡Excelente!

Si lo pruebas ahora, verás que falla como cabe esperar:

```
curl: (7) Failed to connect to localhost port 8000 after 2254 ms: Connection refused
```

## Pues pongámonos a implementarlo

¡Genial!

## Pero una cosa... ¿las APIs REST no se definen en [OpenAPI](#)?

**Preferiblemente, sí.**

En este caso, las describimos en la propia tarea, mediante *markdown*. ¡No siempre tendremos disponible una especificación formal en [OpenAPI](#) de las APIs REST contra las que trabajemos! (Pero sí puede ser una contribución interesante redactarla).

Por ahora, ¡centrémonos en implementar el *endpoint*!

### urls.py

Añadamos una nueva línea de *mapeo*:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('v1/health', endpoints.health_check),
    path('v1/users', endpoints.users), # Asumimos que existe la función users
]
```

### endpoints.py

Y, en este fichero, **añadamos** la función `users` :

```
from django.views.decorators.csrf import csrf_exempt # No nos olvidemos del import

# ...

@csrf_exempt
def users(request):
    # ...
```

Predeciblemente no vamos a soportar otra cosa que no sea un **POST** a este *endpoint*:

```
def users(request):
    if request.method != 'POST':
        return JsonResponse({'error': 'HTTP method not supported'}, status=405)
    # ...
```

Ahora, *procesaremos* los valores que esperamos del cuerpo JSON de la petición:

```
import json # No nos olvidemos del import

# ...

@csrf_exempt
def users(request):
    if request.method != 'POST':
        return JsonResponse({'error': 'HTTP method not supported'}, status=405)
    body_json = json.loads(request.body)
    json_username = body_json['username']
    json_email = body_json['email']
    json_password = body_json['password']
```

## ¿Controlamos errores si no viene algún valor en el JSON?

De momento, no. Programemos el *happy path*.

## Entonces, ¿sólo falta guardar el usuario en la base de datos?

¡Sí!

Con nuestros conocimientos de mapeo objeto-relacional, sabemos que basta con hacer algo *como*:

```
# ...
user_object = CustomUser(email=json_email, username=json_username, encrypted_password=json_password)
user_object.save()
```

Pero, ¡ojo!

¿Qué hay de...

```
encrypted_password=json_password
```

¿No queremos *encriptar* la contraseña antes de guardarla?

## Sí, pero, ¿cómo?

Hablemos de...

## Contraseñas y bases de datos

*Rule of thumb*: **Nunca** almacenar las contraseñas de los usuarios *en texto claro* dentro de la base de datos. Nuestra base de datos puede ser atacada y la información robada.

### Una solución: *Hashear* contraseñas

#### ¿Qué es una [función hash](#)?

Una función que *transforma* un *dato de entrada* en un *valor de salida*. Existen varias funciones *hash* populares como [MD5](#) ó [SHA256](#).

Para comprender cómo funcionan, mejor expliquemos una función *hash* más sencilla: **Función de paridad**.

Todos los números naturales son **pares** ó **impares**. Digamos que la **función de paridad** *transforma* cada número en un **0** ó **1**, dependiendo de si es *par* ó *impar*.

N	f(N)
0	0
1	1
4	0
17	1
18	0

Podemos ver la *función de paridad* como un ejemplo sencillo de función *hash*.

#### ¿Qué [propiedades](#) ha de tener una buena función *hash*?

- Determinista: Para una entrada **N**, siempre produce la misma salida **f(N)**
- Computacionalmente eficiente: ¡Que el ordenador lo haga rapidito!
- No reversible: No debe poder obtenerse **N** a partir de **f(N)**. Si eso es posible, no nos sirve de nada
- Propiedad de avalanche: *Pequeños* cambios en la entrada **N** deben traducirse en *enormes* cambios en la salida **f(N)**
- Resistente a colisiones: Hablamos de *robustez* de la función *hash*. Tiene que ver con la [paradoja del cumpleaños](#)

#### ¿La *función de paridad* cumple esas propiedades?

¡Para nada! Pero sí otras funciones *hash* como [SHA-1](#):

N	f(N)
hola	99800b85d3383e3a2fb45eb7d0066a4879a9dad0

N	f(N)
ola	793f970c52ded1276b9264c742f19d1888cbaf73
my_password	5eb942810a75ebc850972a89285d570d484c89c4

## Pero aún hay más...

*Hashear* una contraseña antes de almacenarla **no** es suficiente. Existe un ataque denominado [rainbow table attack](#) que *crackea* con sencillez contraseñas guardadas así, mediante el uso de tablas de *hashes* precalculados.

## Pues fin de la partida. Borro mi Facebook y mi Instagram

¡No tan rápido!

Como desarrolladores de la base de datos, podemos usar un *salt*.

### ¿[salt](#)?

Es como llamamos a un *churro aleatorio de bytes* generado en el momento de almacenar la contraseña. Así:

1. El **salt** se concatena a la contraseña en *claro*
2. Se *hashea* la concatenación de **salt+contraseña**
3. Se almacena toda la información en un único campo

¡Listo! Esto imposibilita ataques [rainbow table attack](#), puesto que el **salt** se genera aleatoriamente.

### ¿Y qué función *hash* usaremos nosotros?

[BCrypt](#), un algoritmo de *hash* basado en [Blowfish](#).

¡Bastan dos líneas!

```
import bcrypt

# ...
salted_and_hashed_pass = bcrypt.hashpw(json_password.encode('utf8'), bcrypt.gensalt()).decode('utf8')
```

## Y completamos el código

```
from .models import CustomUser

# ...
user_object = CustomUser(email=json_email, username=json_username, encrypted_password=salted_and_hashed_pass)
user_object.save()
return JsonResponse({"is_registered": True}, status=201)
```

¡Adelante!

Prueba a **registrar** un usuario efectuando un **POST**:

```
curl -X POST http://localhost:8000/v1/users -d '{"email": "testemail@test.email.co.uk", "username": "Bill Doe", "pas
```

¿Tienes éxito?

¿Cómo se ve el campo **encrypted\_password** si inspeccionas la base de datos con **sqlite3.exe**?

## Por último

Verifica que tu código pasa el *test* asociado a la tarea.

Haz **commit** y **push** para subir los cambios al repositorio.



Rubén Montero @ruben.montero changed milestone to [%Sprint 5](#) 3 days ago

