


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

Un endpoint para crear sesiones

Resumen

- Definiremos cómo es la petición REST mediante la que soportaremos el *login* de usuarios
- Crearemos una función en `endpoints.py` que procese dicha petición
- Leerá del cuerpo del POST el *email* y *contraseña*
- Los validará con `checkpw`
- Si la contraseña es correcta, generará un *token* aleatorio
- Lo persistirá en la base de datos y también lo devolverá en la respuesta JSON

Descripción

Nuestro API REST soportará la siguiente operación REST:

`/v1/sessions`

POST

Resumen:

Crea una nueva sesión

Parámetros

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

Ejemplo de cuerpo de petición

```
{
  "login_email": "testemail@test.email.co.uk",
  "password": "asdf1234"
}
```

Respuestas

Código	Descripción
201	La sesión se ha creado con éxito
400	La petición HTTP no se envió con todos los parámetros en el JSON
401	La contraseña es incorrecta
404	El usuario especificado no existe

Ejemplo de cuerpo de respuesta (201)

```
{
  "token": "51a3bc2ada4909f5dffd50e468aa887ca2a0a62b"
}
```

¡Qué bien! ¡Sesiones!

¡Sí!

Pero... ¿cómo funciona exactamente eso de *loguearse*?

Las [APIs REST \(REpresentational State Transfer\)](#) no almacenan estado del cliente.



En algún lugar debe guardarse esa información

La *sesión* del cliente se persiste en la base de datos.

Concepto de sesión: Autenticarse con un token



¡Entiendo!

¡Genial!

Pues pongámonos a implementarlo

urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('v1/health', endpoints.health_check),
    path('v1/users', endpoints.users),
    path('v1/sessions', endpoints.sessions) # Nueva línea
]
```

endpoints.py

Añadamos la función `sessions`, donde, predeciblemente no vamos a soportar otra cosa que no sea un **POST**:

```
@csrf_exempt
def sessions(request):
    if request.method != 'POST':
        return JsonResponse({'error': 'HTTP method unsupported'}, status=405)
    # ...
```

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

```
@csrf_exempt
def sessions(request):
    if request.method != 'POST':
        return JsonResponse({'error': 'HTTP method unsupported'}, status=405)
    body_json = json.loads(request.body)
    json_email = body_json['login_email']
```

```
json_password = body_json['password']
# ...
```

Siguientes pasos...

1. Recuperar al usuario con *email* `json_email` de la base de datos
2. Comprobar que la *contraseña* `json_password` es correcta
3. Si lo es, generar una nueva *sesión* (`UserSession`)
4. Guardarla en la base de datos y devolverla en una respuesta JSON como está especificado

Vayamos poco a poco.

1. Recuperar al usuario con *email* `json_email` de la base de datos

Eso sabemos hacerlo:

```
# ...
try:
    db_user = CustomUser.objects.get(email=json_email)
except CustomUser.DoesNotExist:
    pass # No existe el usuario
```

2. Comprobar que la *contraseña* `json_password` es correcta

Ahora, como lo que tenemos guardado en la base de datos es una *hash* de contraseña y *salt*, **no** podemos simplemente hacer una comparación de igualdad (`==`).

Necesitamos coger el *salt* de la base de datos, concatenarlo a la contraseña, *hashearla*, y, entonces, verificar si el *hash* calculado y el almacenado son iguales.

Todo esto se hace con una sola línea¹:

```
# ...
if bcrypt.checkpw(json_password.encode('utf8'), db_user.encrypted_password.encode('utf8')):
    # json_password y db_user.encrypted_password coinciden
else:
    # No coinciden
    pass
```

3. Generar una nueva *sesión* (`UserSession`)

Ahora bien, cada *sesión* se identifica por un *token* aleatorio.

Aunque podríamos usar la librería `random`, se considera criptográficamente más seguro emplear `secrets`:

```
import secrets
from .models import CustomUser, UserSession

# ...

# json_password y db_user.encrypted_password coinciden
random_token = secrets.token_hex(10)
session = UserSession(creator=db_user, token=random_token)
```

4. Guardarla en la base de datos

```
# ...
session.save()
```

5. Devolverla en una respuesta HTTP como está especificado

```
# ...
return JsonResponse({"token": random_token}, status=201)
```

¡Autenticación lista!

Con este *token* que los clientes deben recordar, nos aseguraremos de que *ellos son* quienes en el futuro envíen las peticiones que requieran autenticación.

¿Nuestro POST funciona?

Prueba a lanzar un `cURL` :

```
curl -X POST http://localhost:8000/v1/sessions -d '{"login_email\": \"testemail@test.email.co.uk\", \"password\": \"asdf1234\"}'
```


¿Tienes éxito?

¿Ves el *token* generado?

Por último

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

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

-
1. `.encode('utf8')` es necesario para transformar texto en *bytes* (aquello con lo que trabajan las funciones criptográficas) 
-



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