HOME TASK TRACKER API

Raúl Fernández Cruz

Proyecto Integrado

API REST realizada con Django REST Framework.

Raúl Fernández Cruz
yanngelmarbella@gmail.com

ÍNDICE

1. DESCRIPCIÓN BÁSICA DEL PROYECTO

Home Task Tracker API es una *API REST* realizada con *Django REST Framework*. Sirve para poder gestionar, en grupo, tareas de la casa. En esta se podrán crear grupos, con otros usuarios (o estando únicamente tú). En estos grupos podremos añadir rutinas, por ejemplo "Compras", "Limpieza", etc., y en cada una de estas rutinas podremos crear tareas.

Se trata de una *API REST*, para que sea lo más escalable posible, con intenciones de desarrollar en futuro su app móvil y su panel web.

2. FINALIDAD DEL PROYECTO

Tras 2 años viviendo con compañeros de piso en lugar de con mis padres, me he dado cuenta de que, a veces, no es tan sencillo como parece organizarse y gestionar quién hace cada tarea, con qué frecuencia hacerla, cómo hacerla... Mi idea con este proyecto, es aportar facilidad y transparencia a los estudiantes y familias que convivan juntos, para facilitar las tareas del hogar a nivel organizativo.

3. DESCRIPCIÓN DETALLADA DEL PROYECTO

1. Herramientas de desarrollo:

Django REST Framework.

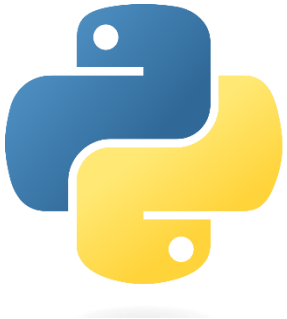
Es un potente *framework* de *Python* usado para la creación de *API REST*.

He elegido este *framework* porque lo he usado en mi experiencia DUAL y en las prácticas, además, considero que además de ser bastante potente, sé manejarlo con bastante soltura.

También creo que se adapta muy bien al proyecto que tenía en mente y es una de las mejores opciones para desarrollar el *back-end*.

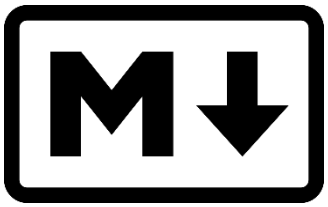
PyCharm

Es el *IDE* que he utilizado para desarrollar el proyecto. Es uno de los *IDE* desarrollado por *JetBrains*, enfocado principalmente al desarrollo usando *Python*. Personalmente, considero que es el mejor *IDE* para programar en este lenguaje, además de que me siento extremadamente cómodo en el mismo.



Python

Python 3.10 concretamente, un lenguaje de programación interpretado, de alto nivel y multifuncional. He elegido la versión 3.10 porque cuando empecé el proyecto era la última *release*. Además, ya que he usado *Django REST Framework*, lógicamente, estoy usando *Python*.

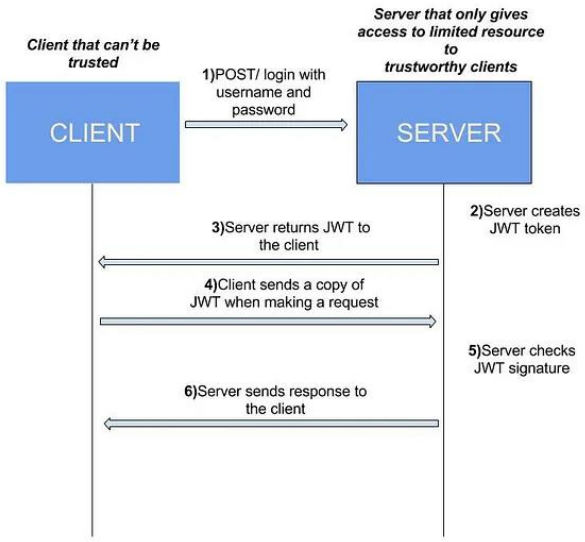


MarkDown

MarkDown es un lenguaje de marcado, que sirve para dar formato a texto plano, siendo este muy sencillo, y usado mundialmente para la documentación de muchos proyectos. Si bien, me hubiera gustado tener más ficheros *.md* (*MarkDown*), y tener todo el proyecto mucho más documentado, los ficheros de documentación en mi proyecto lo son. En el futuro, se incorporarán más.

Swagger

Swagger es un conjunto de herramientas, muy utilizado para las API REST. Lo he configurado en mi proyecto Django REST, para poder documentar los endpoints del proyecto y los métodos que se pueden utilizar en cada uno de estos endpoints. Para ello, he instalado la librería, la he configurado en el settings y he creado en cada uno de los módulos un fichero llamado doc_decorators.py para poder documentar los endpoints de cada módulo, hablaré más a fondo en su apartado correspondiente.



Allauth, dj-rest-auth y SimpleJWT

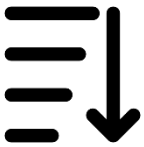
La autenticación es uno de los apartados que más complejidad han añadido a mi proyecto. Para ella he usado 3 librerías que son compatibles y de hecho para algunas funciones necesitas varias de ellas simultáneamente. *Allauth* y *dj-rest-auth* son librerías que sirven para añadir autenticación y registro a nuestra API. En el apartado de settings del que hablaremos más adelante, se permiten diferentes configuraciones y se puede incluso añadir funcionalidad, ya que el

código está en la web de la documentación, y puedes sobrescribir funciones. Sobre la funcionalidad que uso hablaré en su apartado.

Respecto a *SimpleJWT*, es una librería que sirve para aportar seguridad a nuestro servidor. Esta permite la autenticación y uso de peticiones con tokens. Esto lo haremos añadiendo a nuestras peticiones un header de Authentication, y añadiendo "Bearer [token recibido]". Explicado con detalle más adelante.

Advanced REST Client

Advanced REST Client es un programa de escritorio utilizado para testear y realizar peticiones HTTP personalizadas. En el caso de mi proyecto, lo he utilizado para ir comprobando que lo desarrollado, funciona correctamente.

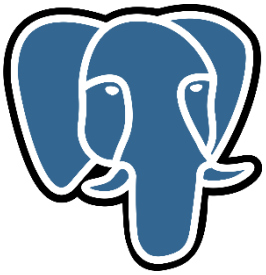


django-filter

django-filter es una librería, común en los proyectos *Django REST Framework*, y se usa básicamente para poder añadirle a las urls indicadores de filtrado y ordenación a los *endpoints* de *GET* que devuelven listados. En este proyecto la he usado en 2 módulos, en el de *Task* y en el de *Group*, para poder filtrar por ciertos parámetros, o para modificar la ordenación. En su apartado explico el "cómo" y el "por qué".

Git y GitHub

Git como herramienta para el control de versiones. GitHub como repositorio. La forma de trabajo ha sido simple, al ser una única persona ha sido muy sencillo y solo he necesitado hacer un mergeo manual por un despiste. Las ramas que he creado han sido "master" (la principal), "base", "development", "group", "routine", "task" y "fixes_and_last_details". (ver en mi GitHub)

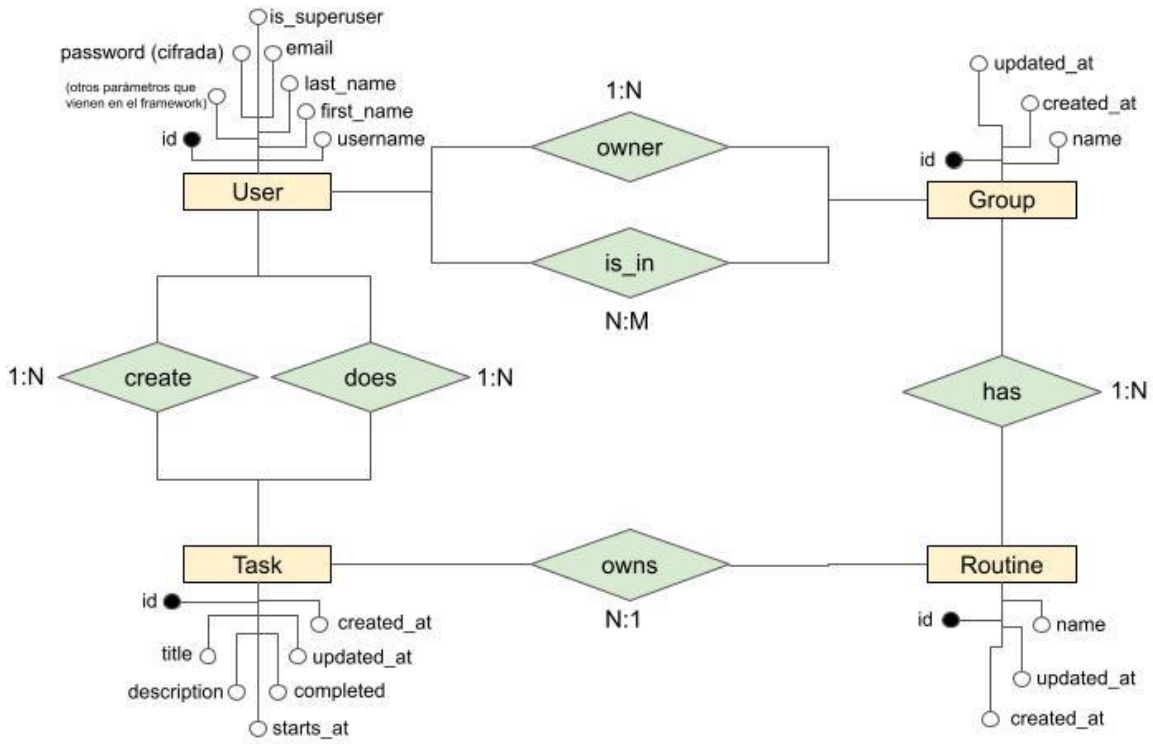


PostgreSQL y pgAdmin

He usado esta tecnología para la base de datos. Ha sido una integración sencilla, ya que el propio *framework* tiene una integración muy sencilla con este sistema gestor.

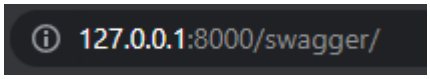
2. Estructura de los datos

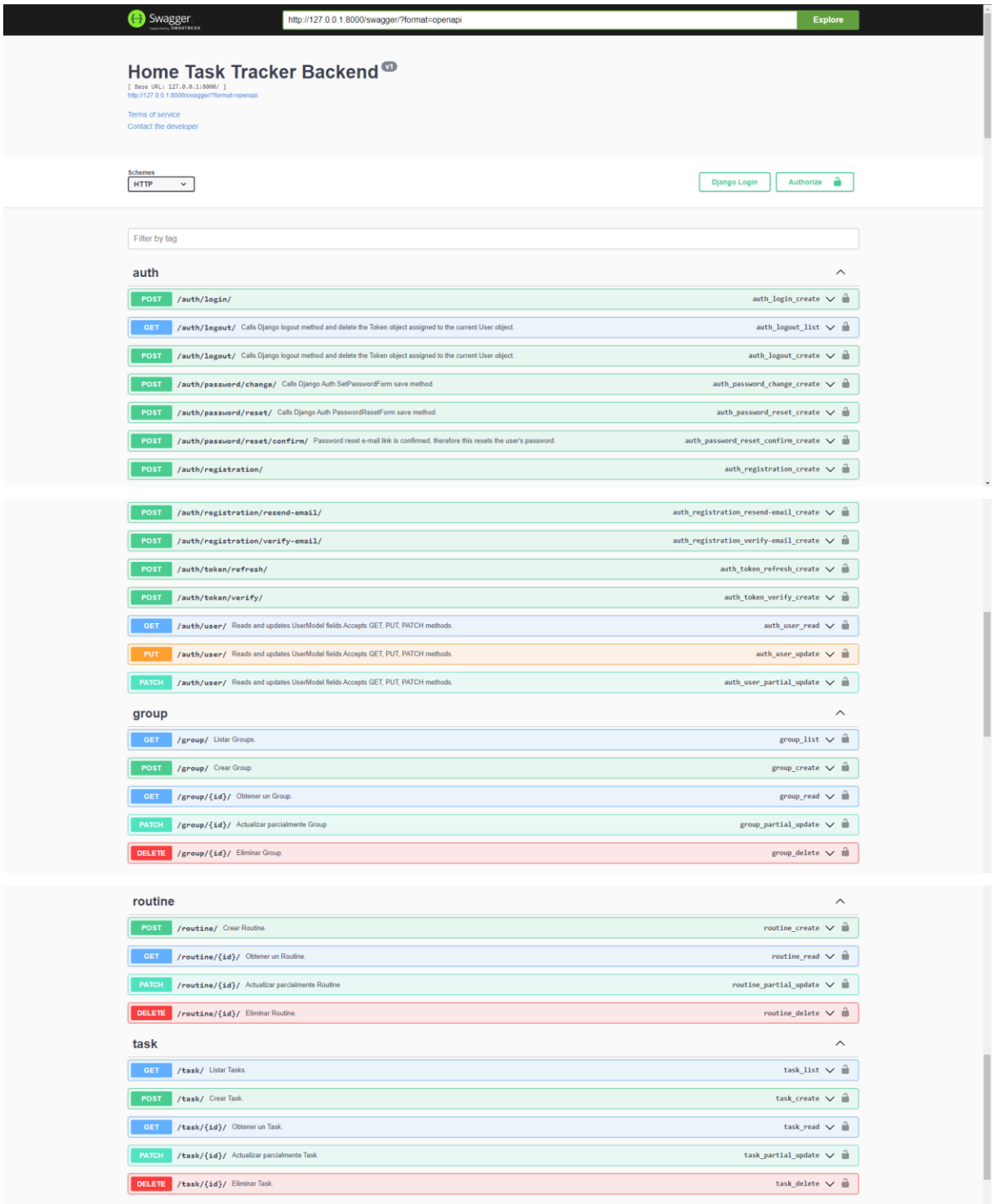
Vamos a hablar de cómo están estructurados. Tenemos una entidad *User*, siendo esta el usuario genérico del framework. Tenemos otra entidad llamada *Group*, la cuál tendrá por un lado un campo "*user_owner*" y otro campo "*user_list*". El campo *user_owner* se añadirá automáticamente, al realizar la petición de crear, con un campo de solo lectura en el serializador de *Group*. Por otro lado, el campo *user_list*, estará compuesto por este *user_owner*, añadido automáticamente, y por los demás usuarios que se manden en la petición. La siguiente entidad es *Routines*, la cuál pertenecerá a un *Group*. Un *Group* podrá tener varias *Routines*, pero una *Routine* solamente podrá pertenecer a un único *Group.* La idea con esto, es poder separar las diferentes rutinas de tareas; por ejemplo, un grupo de usuarios podría crear una rutina para la limpieza, otra rutina para las compras comunes, otra para hacer quedadas o planes en común, además de todo lo que se le ocurra al grupo de usuarios. Al principio, una *Routine* vendrá vacía, ahí es dónde los usuarios crearán objetos *Task*, que además es una entidad relacionada con *Routine*. Esta entidad tendrá los campos mostrados a continuación en el modelo Entidad/Relación, además de 2 relaciones con usuario. ¿Por qué esas relaciones con usuario? Lo he planteado para, por un lado, tener el usuario que ha creado la tarea, el cuál se asignará automáticamente, y, por otro lado, el usuario que la realizará. Ambos pueden ser null, ya que un usuario podría borrar su cuenta, y no por ello queremos que se borre la tarea, ni la asociación con este. Además, creo que tendría sentido crear una tarea para más adelante ver quién la realizará, entonces simplemente habría que actualizar la misma. Abajo dejo el modelo Entidad/Relación que ayudará a entender la explicación de mis decisiones.



3. Endpoints de la API

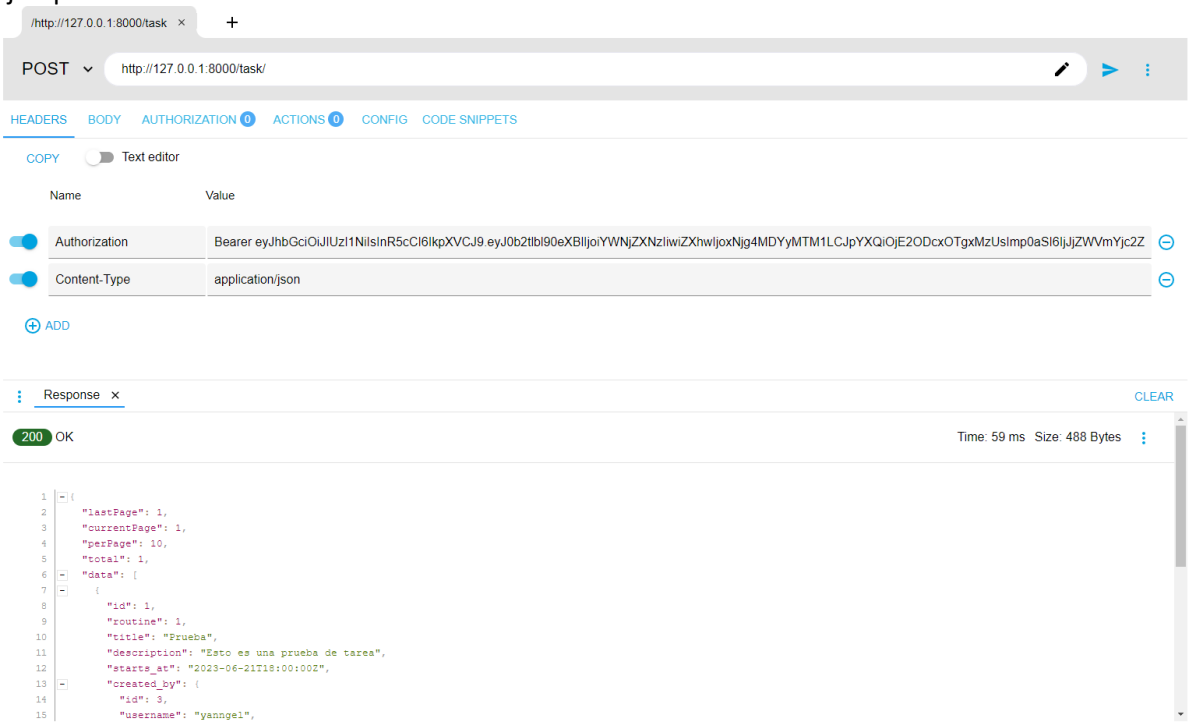
Los endpoints de la API están documentados en el swagger. Para ello accedemos a la url de swagger en el servidor con un navegador web.





En el apartado de “auth” tenemos todas las peticiones relacionadas con los usuarios y acciones que podemos hacer con ellos. Voy a explicar, comenzando desde arriba, cada una de las urls que he definido en el proyecto.

- **/auth/login/ (POST):**
  Recibe un diccionario de datos con los parámetros “username” o “email” y “password”, si el username o el email coinciden con el password, devuelve un token de acceso y un token de refresco. Estos se usarán para realizar las demás peticiones como header de Authorization. Ejemplo:



- **/auth/logout/ (GET):**

  Deshabilita el token de acceso de tu usuario, por lo que tendrás que volver a hacer login para seguir usando la API.

- **/auth/logout/ (POST):**

  Deshabilita el token de acceso de tu usuario, por lo que tendrás que volver a hacer login para seguir usando la API. No se le envían parámetros.

- **/auth/password/change/ (POST):**

  Recibe un diccionario con los parámetros “*password1*” y “*password2*”. Si coinciden será tu nueva contraseña de acceso.

- **/auth/password/reset/ (POST)*:**

  No es funcional actualmente, ya que aún no he configurado el servidor de correo.

  Esta *url* recibirá un diccionario de datos, con un parámetro llamado “email”. El servidor de correo mandará un formulario dónde pondrás la contraseña actual. Si es correcta, devolverá el *uid* de tu usuario, y un *token*. Estos se usarán en la siguiente petición.

- **/auth/password/reset/confirm/ (POST)*:**

  Al igual que la anterior, no es funcional ya que se necesita el servidor de correo.

Recibirá un diccionario de datos con los parámetros new_password1, new_password2, uid y token. Las contraseñas deben coincidir, y el *uid* y el *token* han de ser correctos.

- **/auth/registration/ (POST):**

  Recibirá un diccionario con los campos "*username*", "*email*", "*password1*" y "*password2*". Verifica que el nombre sea único, el mail tenga un formato correcto, las contraseñas sean seguras y además coincidan.

- **/auth/registration/resend-email/ (POST)*:**

  Para cuando el servidor de correos esté configurado con el proyecto, y haya que activar la cuenta con un correo recibido, esta *url* tendrá la función de enviar de nuevo un correo para activar la cuenta.

- **/auth/registration/verify-email/ (POST)*:**

  Se envía un diccionario de datos con un parámetro "*key*", en la cuál se envía la *key* recibida en el email, si es correcta la cuenta se activa.

- **/auth/token/verify/ (POST):**

  Se envía un diccionario de datos con un parámetro "*token*", si este parámetro es el mismo que el token de acceso te

- **/auth/user/ (GET):**

  Recibe los datos principales de tu usuario (pk, username, email, first_name, last_name). Estos datos se declaran en el serializador del usuario.

- **/auth/user/ (PUT):**

  Petición para actualizar tu usuario. Se envía un diccionario de datos con "*username*", "*first_name*" y "*last_name*".

- **/auth/user/ (PATCH):**

  Petición para actualizar parcialmente tu usuario. Se envía un diccionario datos a elegir entre "*username*", "*first_name*" y "*last_name*".

Todas las peticiones que tienen un "", son las que no están operativas, por el mismo motivo, el servidor de correos no configurado.*

Ahora tenemos el apartado de *Group*, *Routine* y *Task*. Todas tienen una estructura similar, ya que son los datos que manejarán los usuarios, y debe ser coherentes para cuando llegue el momento del desarrollo del *front*, los módulos tengan coherencia.

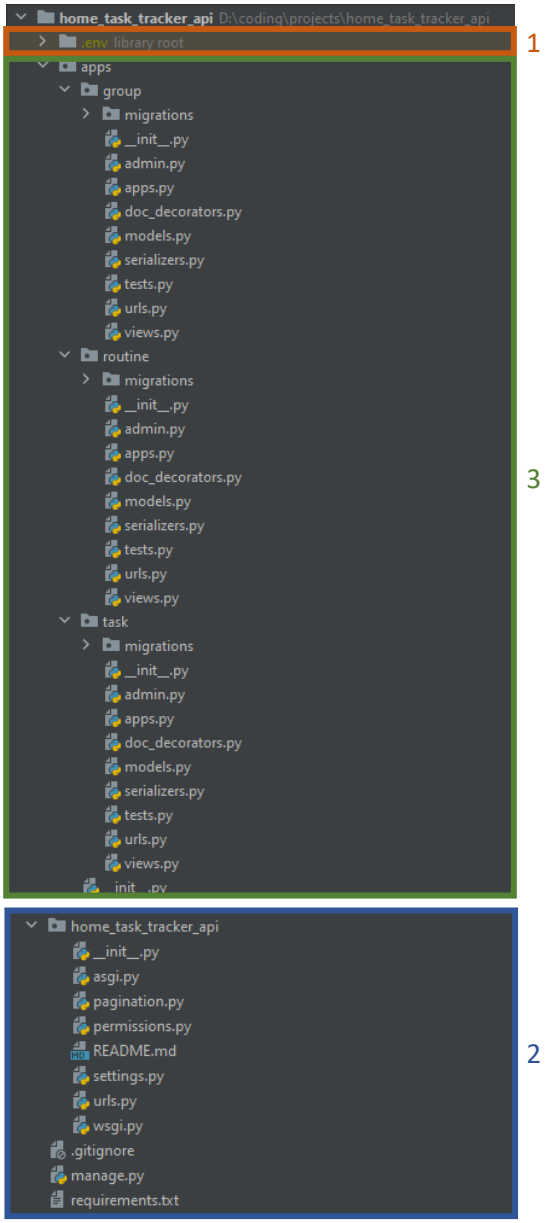
Respecto los datos que se envían y reciben, los mencionaré en el siguiente apartado, en la parte de los serializadores, ya que se enviarán y recibirán los datos definidos en estos.

Todos tienen la misma estructura. /[módulo]/ y /[módulo]/<int:pk>/. Esto quiere decir, que podemos trabajar por un lado con peticiones HTTP GET y POST a /[módulo]/ y por otro lado con peticiones HTTP GET, PATCH y DELETE a las *url* /[módulo]/<int:pk>/, poniendo en <int:pk> la clave primaria (id) de el módulo en concreto. En cuanto al método GET en forma de listado en las *url*, /[módulo]/, lo he permitido en Group y en Task, sin embargo, no en Routine. Esto lo hago ya que, en la aplicación, accederás a un grupo, y dentro de ese grupo verás las diferentes rutinas. Por lo que, listar un conjunto de Routines pienso que no tiene sentido, ya que, si por ejemplo perteneces a "Grupo 1" y "Grupo 2", generarías un listado de rutinas de diferentes grupos, y la idea es que accedas a un grupo en concreto, y dentro de ese grupo, puedas ver las diferentes rutinas que tienen.

Por otro lado, en cuanto al listado de Tasks, sí que lo he permitido, porque he pensado que sería interesante el hecho de las tareas que tienes que realizar.

Todo el funcionamiento lo explico en el apartado a continuación.

4. Descomposición modular y descripción de los módulos.

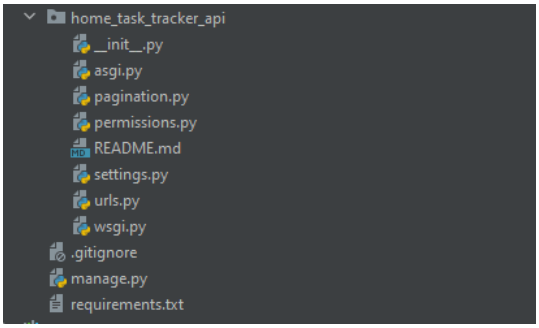


1. *Entorno Virtual*

Respecto al Entorno Virtual, no hay mucho que decir. Es común en Python utilizar un "*virtualenv*" o entorno virtual, para que cada vez que se instale una librería con "*pip install [librería]*" no se guarde en tu sistema, sino que se guarde en el propio entorno virtual del proyecto. Se crea escribiendo en la terminal "*python3 -m venv [nombre del entorno, en mi caso, .env]*"

Para activar el entorno, en Windows, en la raíz del proyecto, se usa el comando "*.env/Scripts/activate/*".

2. *Módulo de configuración y archivos en la raíz*



Este módulo, es donde una vez generado el archivo *.env* y activado, trabajaremos. Sobre los diferentes ficheros que podemos ver aquí, los dos menos relevantes para mi proyecto son los llamados "*asgi.py*" y "*wsgi.py*". Estos sirven básicamente para la configuración de la interfaz web y para la comunicación asíncrona entre servidores en caso de estar trabajando con varios. Estos ficheros no los he modificado, ya que no ha sido necesario en este proyecto, al menos de momento.

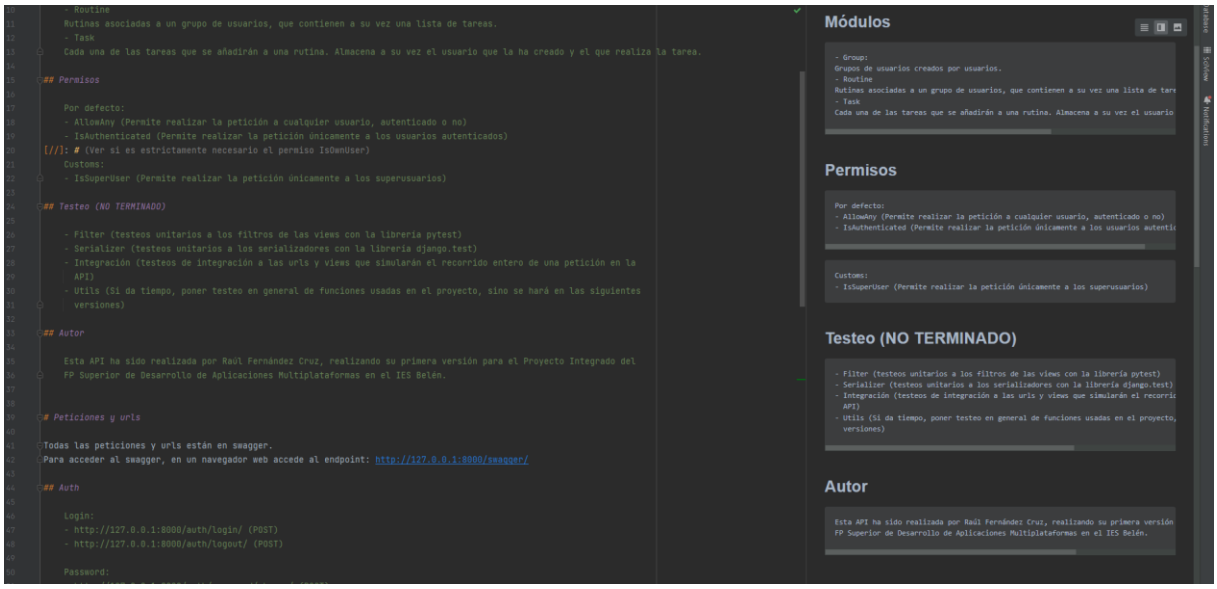
El fichero manage.py, es otro que tampoco he modificado, sin embargo es realmente importante, ya que es el usado para la herramienta de comandos de Django REST, como por ejemplo, algunos comandos de interés, pueden ser "*python manage.py makemigrations*", para realizar las migraciones en el proyecto, "*python manage.py migrate*" para pasar esas migraciones a la base de datos vinculada o "*python manage.py runserver*" para lanzar el servidor (por defecto en el puerto 8000 de localhost 127.0.0.1:8000).

El fichero *.gitignore*, el fichero *requirements.txt* que están en la raíz son ficheros que he añadido a mano, al igual que el fichero *pagination.py*, *permissions.py* o el *README.md*, que se encuentran en la carpeta *home_task_tracker_api* en el módulo de configuración del proyecto. Cada uno tiene funciones diferentes, por ejemplo, el fichero *.gitignore* lo he obtenido combinando el *.gitignore* recomendado por *PyCharm* para proyectos *Django REST* con el recomendado de *GitHub*. Fue el primer fichero que añadí al proyecto, para evitar subir a GitHub cualquier tipo de información que no convenga, como ficheros del *IDE*, o el propio entorno virtual. El fichero *requirements.txt*, es un fichero que he ido generando de la siguiente manera: El comando "*pip freeze*", genera un listado de todas las librerías instaladas, en caso de estar en el entorno virtual, pues en este, entonces, utilizando el comando "*pip freeze > requirements.txt*" generamos un fichero con todas las dependencias en cuánto a librerías se refiere de nuestro proyecto. Si en un futuro quiero trabajar en otro equipo, o alguien más empieza a trabajar en el proyecto, sería tan sencillo como utilizar la versión 3.10 de Python, y una vez generado un entorno virtual, lanzar el comando "*pip install -r requirements.txt*".

```
asgiref==3.6.0
certifi==2022.12.7
cffi==1.15.1
charset-normalizer==3.1.0
coreapi==2.3.3
coreschema==0.0.4
cryptography==40.0.2
defusedxml==0.7.1
dj-rest-auth==3.0.0
Django==4.2.1
django-allauth==0.52.0
django-filter==23.2
djangorestframework==3.14.0
djangorestframework-simplejwt==5.2.2
drf-yasg==1.21.5
idna==3.4
inflection==0.5.1
itypes==1.2.0
Jinja2==3.1.2
MarkupSafe==2.1.2
oauthlib==3.2.2
openapi-codec==1.3.2
packaging==23.1
psycopg2==2.9.6
pycparser==2.21
PyJWT==2.6.0
python3-openid==3.2.0
pytz==2023.3
requests==2.30.0
requests-oauthlib==1.3.1
rest-framework-simplejwt==0.0.2
ruamel.yaml==0.17.22
ruamel.yaml.clib==0.2.7
simplejson==3.19.1
sqlparse==0.4.4
tzdata==2023.3
uritemplate==4.1.1
urllib3==2.0.2
```

Aquí una captura de cómo se ve el fichero *requirements.txt* en mi proyecto.

De los ficheros mencionados, nos queda por una parte el *README.md*, en el cual hay documentación básica del proyecto en *MarkDown*.



(así se ve un fragmento del README.md, puedes verlo entero en mi GitHub)

Por otro lado, el fichero pagination.py básicamente tiene una clase personalizada para modificar el comportamiento de la paginación en el proyecto.

El permissions.py contiene los permisos personalizados, que por ahora solamente he necesitado 1. Lo he llamado *IsSuperUser*, y funciona así:

```python
from rest_framework import permissions


class IsSuperUser(permissions.BasePermission):
    """
    Permiso custom para identificar que es superusuario.
    """
    def has_permission(self, request, view):
        # User
        user = request.user

        # Si existe el usuario y está autenticado, es superusuario
        if user and user.is_authenticated:
            return user.is_superuser

        return False
```

Básicamente, es una clase llamada como el nombre del permiso que quiero que tenga. Esta clase, hereda de *permissions.BasePermission*, el permiso básico. En *Django REST* los permisos funcionan devolviendo *True* o *False*. Entonces, obtengo de la *request* el usuario, y compruebo que existe, y además está autenticado. En caso de que así sea, devuelve si es *superuser* o no. Por lo que, si lo es, devuelve *True*, y si no lo es, devuelve *False*.

Si no se cumple la condición, es decir, si el usuario no existe, o si no ha sido autenticado, devolverá False.

En este módulo, nos quedan los dos ficheros que más he tenido que modificar según avanzaba el proyecto, ya que uno es el de las urls (*urls.py*), y otro es el de la configuración (*settings.py*).

El *settings.py* es un fichero bastante complejo, y en este documento voy a mostrar las partes más relevantes.

```python
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # auth
    'dj_rest_auth',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'dj_rest_auth.registration',

    # Installed apps
    'rest_framework',
    'rest_framework_simplejwt',
    'django_filters',
    'rest_framework.authtoken',

    # Docs
    'drf_yasg',

    # User defined apps
    'apps.group',
    'apps.routine',
    'apps.task',
)
```

En la variable *INSTALLED_APPS*, se añade una tupla, con las diferentes librerías que se usan en este proyecto. Las primeras que aparecen son las que vienen definidas por el propio *framework*. Luego son las que he añadido de autenticación. Seguidamente las librerías de *Django REST*, también instaladas a mano. Luego las de documentación, que únicamente está "*drf_yasg*" (es la librería de *Swagger*). Y para finalizar, en "*User defined apps*" se añaden los módulos que generamos en el proyecto. Para ello, se utiliza el comando "*python manage.py startapp [nombre del módulo]*".

```python
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(days=10),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=50),
    "UPDATE_LAST_LOGIN": True,
}

REST_AUTH = {
    'USE_JWT': True,
    'JWT_AUTH_HTTPONLY': False,
    'REGISTER_SERIALIZER': 'dj_rest_auth.registration.serializers.RegisterSerializer',
    'REGISTER_PERMISSION_CLASSES': ('rest_framework.permissions.AllowAny',),
}

AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    "django.contrib.auth.backends.ModelBackend",
    # `allauth` specific authentication methods, such as login by e-mail
    "allauth.account.auth_backends.AuthenticationBackend"
)

ACCOUNT_AUTHENTICATION_METHOD = 'username_email'
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_UNIQUE_EMAIL = True
ACCOUNT_EMAIL_VERIFICATION = None
```

Respecto a esta parte, aquí tenemos parte de la configuración de la autenticación. En la variable *SIMPLE_JWT* se almacena un diccionario de datos, en los cuales he modificado la configuración por defecto de el token de acceso y de refresco. En este caso le he puesto 10 días al token de acceso y 50 al de refresco, ya que esta es la versión de desarrollo, si hago *deploy* para trabajar en un entorno real, lógicamente reduciría los tiempos de vida de los *tokens*.

En *REST_AUTH*, activo el *JWT (Json Web Token)*, y añado serializadores que encontré en la documentación para que tengan el funcionamiento que deseo.

En *AUTHENTICATION_BACKENDS* configuro que sea posible la autenticación en *Django admin* mediante *username*, y permito también la autenticación mediante email.

Las variables siguientes, en orden, permiten la autenticación en la *url* de *login* mediante nombre de usuario o email. Seguidamente, obligo a la hora de crear una cuenta que se añada el campo de email. Especifico que sea único. La última variable indica que no está activado la verificación por mail, ya que aún no está configurado el servidor de correo.

```python
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
        'rest_framework.filters.SearchFilter',
    ],
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '5/minute',
    },
    'DEFAULT_PAGINATION_CLASS': 'home_task_tracker_api.pagination.CustomPagination',
    'PAGE_SIZE': 10,
}
```

En esta variable del *framework*, vienen algunas cosas por defecto. He tenido que modificar el parámetro de "*DEFAULT_AUTHENTICATION_CLASSES*", "*DEFAULT_FILTER_BACKENDS*" y "*DEFAULT_PAGINATION_CLASS*".

```python
# Swagger documentation SETTINGS.
# Package is drf_yasg
SWAGGER_SETTINGS = {
    'DEFAULT_INFO': 'home_task_tracker_api.urls.api_info',
    'SECURITY_DEFINITIONS': {
        'Bearer': {
            'type': 'apiKey',
            'name': 'Authorization',
            'in': 'header'
        }
    }
}
```

La definición del *Swagger* la he añadido al settings tal cual viene en la documentación.

```python
# Database
# https://docs.djangoproject.com/en/4.2/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': os.environ.get('POSTGRES_DB', "htt_local_db"),
        'USER': os.environ.get('POSTGRES_USER', "user_htt"),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD', "password"),
        'HOST': os.environ.get('POSTGRES_HOST', "127.0.0.1"),
        'PORT': os.environ.get('POSTGRES_PORT', "5432"),
        'TEST': {
            'NAME': 'test_pgdb'
        }
    }
}
```

En este apartado del *settings*, configuro la base de datos. Por defecto, viene con una configuración para una base de datos *sqlite* en la raíz del proyecto, pero se puede modificar para tenerlo en una base de datos diferente. La seguridad no es potente, ya que al final, recordemos que este es el proyecto en local y en *debug*, una vez deployeado tendría otra configuración específica para este caso en concreto.

El último apartado que considero interesante en el *settings*, es el *AUTH_PASSWORD_VALIDATORS*, el cual he dejado con algunos validadores genéricos para la contraseña.

```python
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

En el fichero *urls.py*, tenemos todo mucho más claro. Al principio declaro 2 variables para condensar en ellas toda la autenticación. Estas son *dj_rest_auth_path* y *dj_rest_auth_path_registration*. En ambas incluyo las url por defecto de las librerías elegidas para la autenticación.

Además, he creado 2 variables, una llamada *api_info*, en la cual creo un objeto *Info* utilizando la librería *openapi*. Esta variable, la añado en el constructor del *schema*, el cual sirve para más alante utilizar como vista para la *url* del *Swagger*.

En la variable *urlpatterns* defino todas las *urls* del proyecto. Añado las previamente declaradas, y lo mismo con la del swagger, la cual hay que además añadirle el parámetro .with_ui y especificarle algunos parámetros.

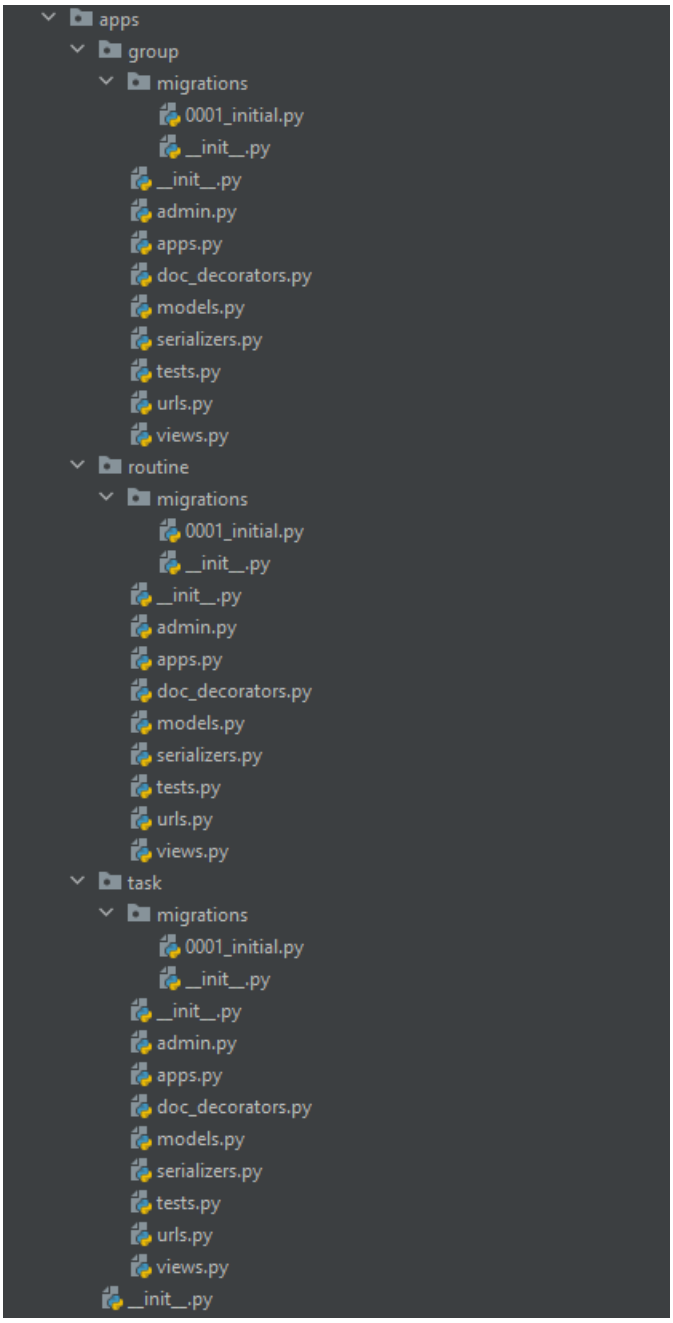
```python
from django.contrib import admin
from django.urls import path, include
from drf_yasg import openapi
from drf_yasg.views import get_schema_view
from rest_framework import permissions

# AUTH (https://dj-rest-auth.readthedocs.io/en/latest/introduction.html)
dj_rest_auth_path = path('auth/', include('dj_rest_auth.urls'))
dj_rest_auth_path_registration = path('auth/registration/', include('dj_rest_auth.registration.urls'))
# TODO ¿social authentication? ej. Iniciar Sesión con Google

# SWAGGER (https://django-rest-swagger.readthedocs.io/en/latest/)
api_info = openapi.Info(
    title="Home Task Tracker Backend",
    default_version='v1',
    terms_of_service="TODO",
    contact=openapi.Contact(email="yanngelmarbella@gmail.com"),
)
schema_view = get_schema_view(
    info=api_info,
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    dj_rest_auth_path,
    dj_rest_auth_path_registration,
    path('admin/', admin.site.urls),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
    path('group/', include('apps.group.urls')),
    path('routine/', include('apps.routine.urls')),
    path('task/', include('apps.task.urls')),
]
```

Las demás *urls*, son las de *admin*, en las cuales añado las de la librería de django de admin, que en el siguiente apartado explicaré como he configurado.

Las url de *group*, *routine* y *task*, incluyen las definidas en sus módulos, en el fichero urls. Estos los veremos en el siguiente apartado, cuando veamos cómo he desarrollado los módulos.

El *# TODO (por hacer)*, indica que quería añadir "*social authentication*", para poder iniciar sesión directamente con tu cuenta de *Google*. Es algo que tengo intenciones de añadir de cara a futuro en el proyecto.

3. *Fichero apps con los módulos de Django REST*



En cada módulo podemos apreciar que hay una carpeta para las migraciones, las cuales son incrementales y sirven para cada vez que lanzamos las migraciones, si se ha modificado un modelo dentro del módulo (o se ha creado).

Los modelos son las clases que se comunican más directamente con la base de datos.

Las views son las funciones que tienen las urls, es decir, por ejemplo, un *GET* llama a una función de una *view*. A su vez, las *views* están comunicadas con los serializadores, ya que, si una *url* de *POST* recibe una serie de datos, la view accederá al serializador para que valide y guarde los datos en el modelo.

Los *serializers* son ficheros que sirven para validar los datos recibidos, además de comunicar las *views* con los modelos, y viceversa. Por ejemplo, una *view* de obtener datos, llamaría al serializador para poder mostrar los mismos con el formato especificado en el serializador. Al revés sucede lo mismo, para guardar datos, la vista hace uso del serializador para validar los datos, y una vez validados, guardarlos en el modelo.

El fichero *admin.py* sirve para configurar la *url* de *admin* de nuestro servidor.

El fichero *apps.py* sirve para darle un nombre al módulo, para declarar este seguidamente en el *settings*, tal y como vimos anteriormente.

Finalmente, el fichero *doc_decorators* he creado funciones para cada una de las views. Estas funciones, son las encargadas de mostrar en la *url* del *Swagger* la información de las peticiones que hemos creado.

Ahora, voy a mostrar el código de cada una de estas partes en el módulo *Task*, y explicar las funcionalidades más relevantes.

```python
9 usages
class Task(models.Model):
    """
    Modelo Task
    - id: (IntegerField) Número entero, único y auto-incremental, asignado automáticamente.
    - routine: (ForeignKey) Rutina a la que pertenece la tarea.
    - title: (CharField) Título de la tarea que se realizará.
    - description: (TextField) Campo de texto no obligatorio que dará una descripción sobre lo que se realizará.
    - starts_at: (DateTimeField) Fecha y hora a la que comienza la tarea.
    - created_by: (ForeignKey) Usuario que ha creado la tarea (se asignará automáticamente mediante el serializador).
    - assigned_to: (ForeignKey) Usuario que realizará la tarea.
    - completed: (BooleanField) Booleano que indica si la tarea está terminada o no.

    - created_at: (DateTimeField) Fecha en la cual se creó el grupo.
    - updated_at: (DateTimeField) Fecha de la última actualización realizada al grupo.
    """
    routine = models.ForeignKey(Routine, verbose_name='Rutina', on_delete=models.CASCADE, related_name='tasks')
    title = models.CharField(verbose_name='Título', max_length=100)
    description = models.TextField(verbose_name='Descripción', max_length=2048, null=True, blank=True)
    starts_at = models.DateTimeField(verbose_name='Fecha y hora de inicio')
    created_by = models.ForeignKey(User,
                                   verbose_name='Usuario que ha creado la tarea',
                                   null=True,
                                   related_name='tasks_created',
                                   on_delete=models.SET_NULL)
    assigned_to = models.ForeignKey(User,
                                    verbose_name='Usuario que realizará la tarea',
                                    null=True,
                                    related_name='assigned_tasks',
                                    on_delete=models.SET_NULL)
    completed = models.BooleanField(verbose_name='Completada', default=False)

    created_at = models.DateTimeField(verbose_name='Fecha de creación', auto_now_add=True)
    updated_at = models.DateTimeField(verbose_name='Fecha de actualización', auto_now=True)

    def __str__(self):
        return f'Task [{self.pk}] {self.title}'

```

Este es el modelo de *Task*. Como vemos, hereda de *models.Model*. En la documentación explico algo más a fondo cada campo. Podemos ver que cada una de las variables que declaro, salen de "*models.[tipo]*". Además, añado parámetros para que tengan el funcionamiento necesario. El método *__str__* es un ´metodo estático de la clase, y sirve para mostrar los objetos cuando se muestren en una terminal para hacer *debug*, o en la pantalla de *admin* del proyecto. Los parámetros *verbose_name*, sirven para básicamente lo mismo, en lugar de salir tal cual en la pantalla de *admin*, aparecerán con el nombre que hemos definido.

```python
10 usages
class TaskSerializer(serializers.ModelSerializer):
    """
    Serializer de Task

    - id: (IntegerField) Número entero, único y auto-incremental, asignado automáticamente.
    - routine: (ForeignKey) Rutina a la que pertenece la tarea.
    - title: (CharField) Título de la tarea que se realizará.
    - description: (TextField) Campo de texto no obligatorio que dará una descripción sobre lo que se realizará.
    - starts_at: (DateTimeField) Fecha y hora a la que comienza la tarea.
    - created_by: (ForeignKey) Usuario que ha creado la tarea (se asignará automáticamente mediante el serializador).
    - assigned_to: (ForeignKey) Usuario que realizará la tarea.
    - completed: (BooleanField) Booleano que indica si la tarea está terminada o no.

    - created_at: (DateTimeField) Fecha en la cual se creó el grupo.
    - updated_at: (DateTimeField) Fecha de la última actualización realizada al grupo.
    """
    created_by = UserTaskSerializer(read_only=True)
    assigned_to = UserPKRelatedField(
        queryset=User.objects.all(),
        required=True,
    )

    class Meta:
        model = Task
        fields = (
            'id',
            'routine',
            'title',
            'description',
            'starts_at',
            'created_by',
            'assigned_to',
            'completed',
            'created_at',
            'updated_at',
        )

        read_only_fields = (
            'id',
            'created_by',
            'created_at',
            'updated_at',
        )

    def create(self, validated_data):
        # Obtener el usuario de la petición.
        user = self.context['request'].user

        # Asociarlo al created_by.
        validated_data['created_by'] = user

        return super().create(validated_data)
```

Este es el serializador de *Task*. En una clase interna llamada *Meta*, básicamente he añadido todos los campos que mostrará el serializador, seguidamente los campos que únicamente son de lectura. Cuando no añades nada, por defecto lo deja con la configuración por defecto, o con la definida en el modelo. Para el usuario que hay en *created_by*, únicamente mostraría su *id* si lo dejase por defecto. Para ello, he creado otro serializador más arriba en este fichero, y en lugar de sacar únicamente su id, saco algunos datos del usuario. Algo similar hago con el usuario asignado. Para este, he creado una clase que hereda de *serializers.PrimaryKeyRelatedField* y modificado su comportamiento. A la hora de introducir datos para crear, funciona de igual forma. Pero a la hora de mostrarlos, modifico la función "*to_representation*" para que use el serializador de usuario, en lugar de mostrar un listado de *ids*.

```python
1 usage
class UserPKRelatedField(serializers.PrimaryKeyRelatedField):
    def to_representation(self, value):
        user = User.objects.get(pk=value.pk)
        user_serialized = UserTaskSerializer(user)

        return user_serialized.data
```

```python
2 usages
class TaskView(viewsets.ModelViewSet):
    """
    View de Task
    """
    queryset = Task.objects.all()

    serializer_class = TaskSerializer

    permission_classes = (IsAuthenticated, )

    filterset_fields = {
        'routine': ['exact'],
        'starts_at': ['gte'],
        'created_by': ['exact'],
        'assigned_to': ['exact'],
        'completed': ['exact'],
    }

    search_fields = {
        'title',
        'description',
    }

    ordering_fields = ['created_at', 'updated_at', 'starts_at']

    def get_queryset(self):
        """
        Modificación del queryset
        """
        user = self.request.user

        # Si es superuser recibirá el queryset genérico
        if user.is_superuser:
            return self.queryset

        # Si el método usado es get en la petición de listar, filtrará por las tareas que debe hacer el usuario
        if 'pk' not in self.kwargs and self.request.method == 'GET':
            return Task.objects.filter(assigned_to=user.id)

        # Si no lo es, filtrará por los grupos a los que pertenezca.
        else:
            return Task.objects.filter(routine__group__users_list=user.id)

    @doc_view_task_list
    def list(self, request, *args, **kwargs):
        """
        Listar Tasks.

        Petición para listar Tareas.
        """
        result = super().list(request, *args, **kwargs)
        return result
```

En cuanto a la *view* en *Task*, he heredado de un tipo de *view* llamado *ModelViewSet*. En este, se pueden modificar ciertos campos y funciones para añadir el comportamiento deseado.

Creo que todas las variables indican bastante claro lo que hacen, respecto al *queryset*, he modificado su funcionamiento, sobrescribiendo la función *get_queryset* de la clase padre. El *queryset* son los datos de base de datos que se procesarán con el serializador, entonces, lo que he hecho ha sido que, según el tipo de usuario, te llegue el *queryset* que he dejado por defecto, que te muestra todos los objetos tipo *Task* si eres *superuser*, si estás haciendo una petición de listar, te muestra las tareas cuyo usuario asignado a hacerla es el que ha realizado la petición, y si no, filtra según si estás en el grupo o no.

Tenemos por último una función de "*list*", otra de "*retrieve*", de "*create*", de "*partial_update*" y por último de "*destroy*". Según el método *HTTP* y url que se use, se accederá a una función u otra.

```python
@doc_view_task_list
def list(self, request, *args, **kwargs):
    """
    Listar Tasks.

    Petición para listar Tareas.
    """
```

Todas las peticiones, tienen una cabecera, con una función creada en el fichero *doc_decorators.py*, con información de la petición en concreto. También tienen un comentario con 2 líneas. Toda esta información, se mostrará en *swagger* cuando hagamos clic en el desplegable de la *url* y su método.

La función *doc_view_task_list* en *doc_decorators.py*:

```python
2 usages
def doc_view_task_list(action):
    """
    This decorator purpose is to reuse the auto schema.
    """
    list_param = [
        openapi.Parameter('search', openapi.IN_QUERY,
                          description='Campos de búsqueda: title, description',
                          type=openapi.TYPE_STRING),
        openapi.Parameter('filter', openapi.IN_QUERY,
                          description='Campos de filtrado: routine, starts_at, created_by, assigned_to, completed',
                          type=openapi.TYPE_STRING),
        openapi.Parameter('ordering', openapi.IN_QUERY,
                          description='Campos de ordenación: created_at, updated_at, starts_at',
                          type=openapi.TYPE_STRING)
    ]

    list_responses = {
        401: 'El usuario debe estar autenticado',
        200: openapi.Response('response description', TaskSerializer(many=True)),
    }
    return swagger_auto_schema(manual_parameters=list_param, responses=list_responses)(action)

```

Por último, las *urls* de *Task*. Anteriormente, en las *urls* del proyecto mostré lo siguiente:

```python
    path('task/', include('apps.task.urls')),
```

Esto, quiere decir que a partir del directorio 127.0.0.1:8000/*task/*, añada el funcionamiento que especifique en el directorio del proyecto *apps/taks/urls.py*, el cual es el siguiente:

```python
from django.urls import path
from apps.task import views

urlpatterns = [
    path('', views.TaskView.as_view({'get': 'list', 'post': 'create'}), name='tasks'),
    path('<int:pk>/', views.TaskView.as_view(
        {
            'get': 'retrieve',
            'patch': 'partial_update',
            'delete': 'destroy'
        }
    ), name='tasks_detail'),
]

```

Lo que hago en este fichero, es indicar que en la url */task/*, añada la view mostrada anteriormente, y si hace un método *get*, utilice la función llamada *list*, si hace un método *post* utilice la función *create*.

En la de abajo el funcionamiento es lo mismo. Simplemente, le añado un segundo parámetro a la *url*, que debe ser un número entero, y se enviará como argumento adicional con el nombre de "*pk*". Esto lo he hecho así, para las peticiones que trabajan con un único objeto.

Por último, para poder utilizar el endpoint */admin/*, se crea una clase en el fichero *admin.py*, heredando de la clase *admin.ModelAdmin*.

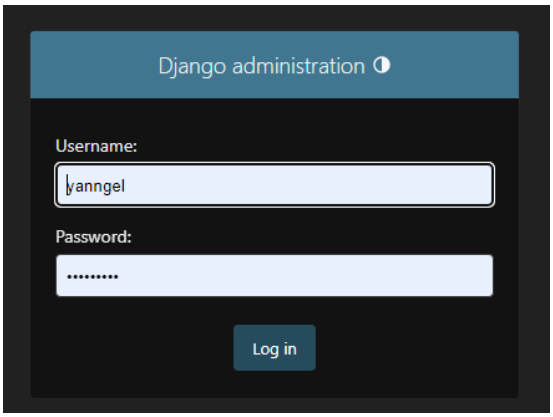
```
2 usages
class TaskAdminConfig(admin.ModelAdmin):
    model = Task

    # Campos de escritura
    fields = (
        'routine',
        'title',
        'description',
        'starts_at',
        'created_by',
        'assigned_to',
        'completed',
    )

    # Campos de búsqueda (en la view)
    search_fields = (
        'title',
        'description',
    )

    # Campos de filtro (en la view)
    list_filter = (
        'routine',
        'starts_at',
        'created_by',
        'assigned_to',
        'completed',
    )

    # Campos que se muestran (todos)
    list_display = (
        'id',
        'routine',
        'title',
        'description',
        'starts_at',
        'created_by',
        'assigned_to',
        'completed',
        'created_at',
        'updated_at',
    )

    # Sobreescribir el form para que el campo de users_list no sea obligatorio
    def get_form(self, request, obj=None, change=False, **kwargs):
        form = super(TaskAdminConfig, self).get_form(request, obj, **kwargs)
        form.base_fields['description'].required = False
        form.base_fields['created_by'].required = False
        form.base_fields['assigned_to'].required = False
        return form


admin.site.register(Task, TaskAdminConfig)
```

En los comentarios explico lo que realiza cada variable y función.

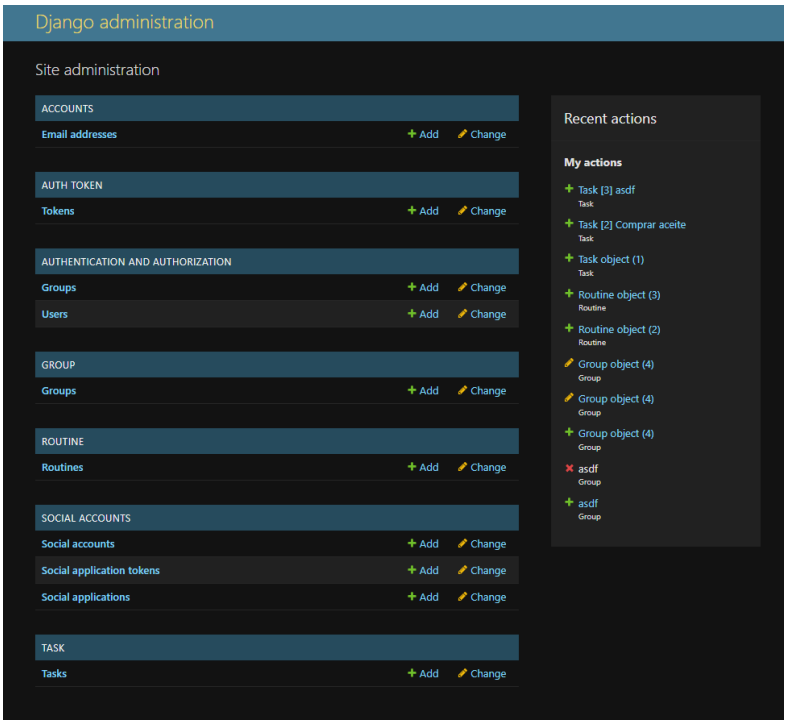
Este mismo estudio y paradigma de programación, lo he realizado en los otros módulos, para así tenerlo todo vinculado y accesible.

5. Sitio de Admin

Esta es la otra zona gráfica además del *Swagger*, sin embargo, en esta se puede trabajar con la *API* directamente con la base de datos como super usuario.

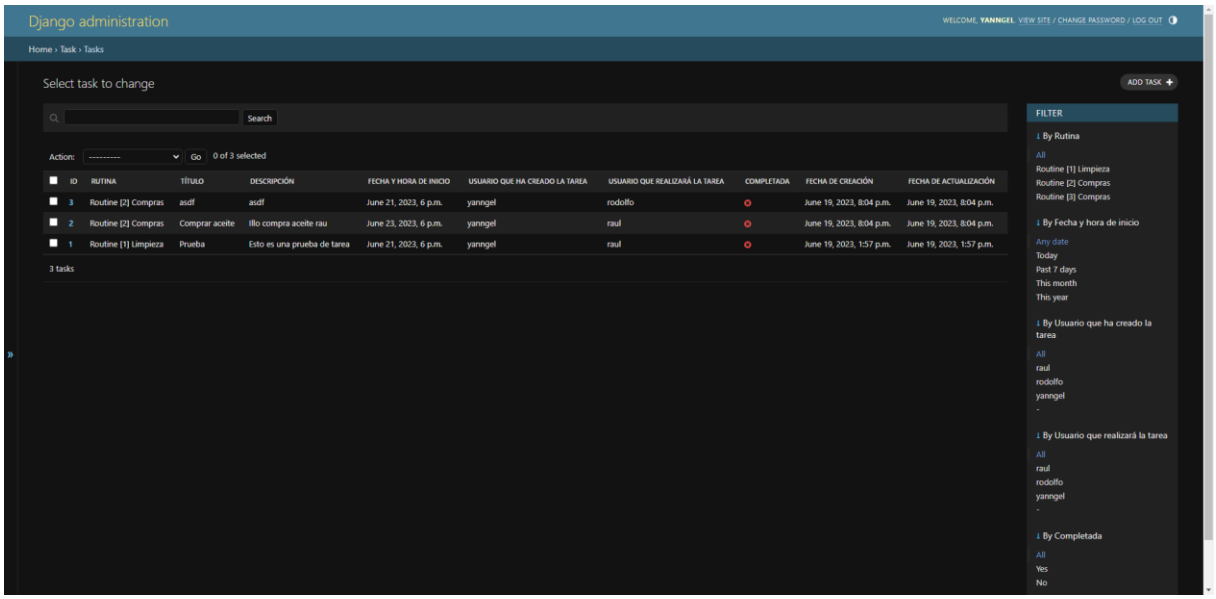


Iniciamos sesión con un super usuario.

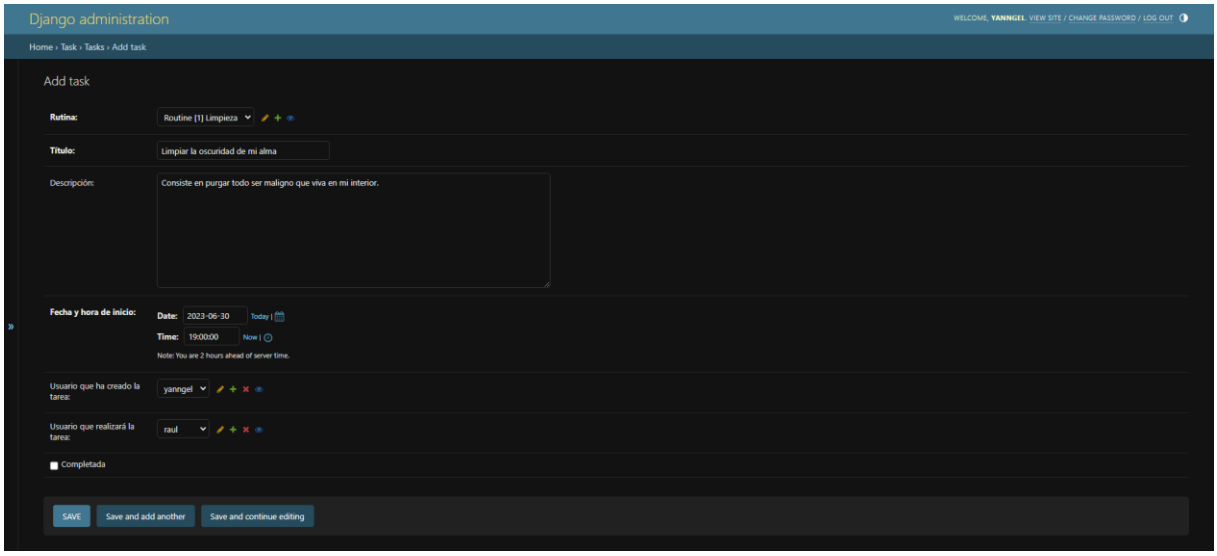


Podemos ver, cómo podemos interactuar directamente con la base de datos mediante este *endpoint* de la *API*. A la derecha hay un log de algunas pruebas que he ido realizando.

Y aquí tenemos la pantalla con las diferentes *Tasks* que tenemos en base de datos. Podemos ordenar por los campos que hemos permitido, buscar en el buscador usando como referencia los campos especificados y también filtrar.



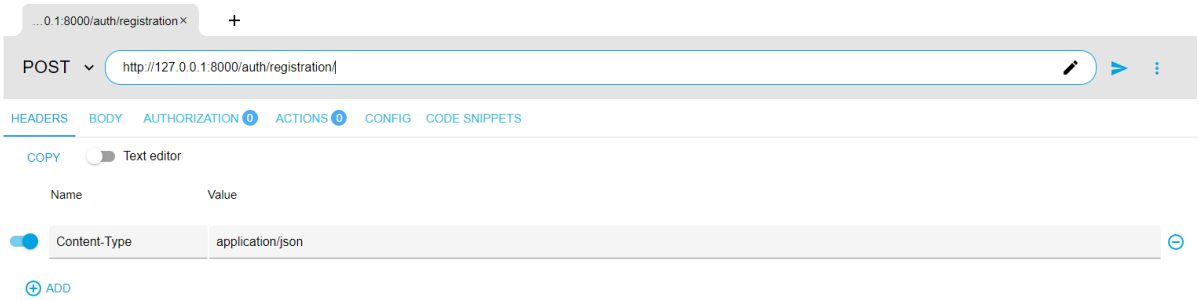
Si clicamos en "*Add Task*", podemos crear una tarea.



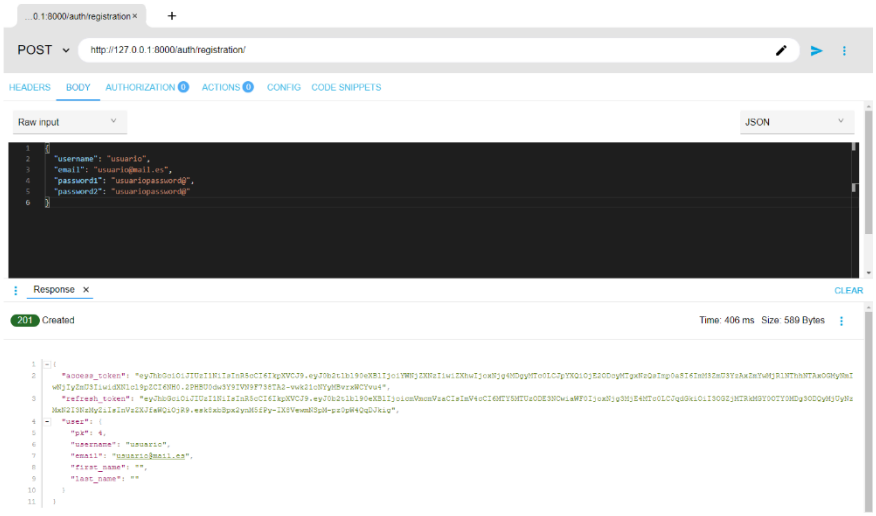
Y podemos crear esta tarea, asignándosela al usuario "raul", que forma parte del grupo el cuál tiene la rutina 1.

6. Funcionamiento para usuarios no administradores

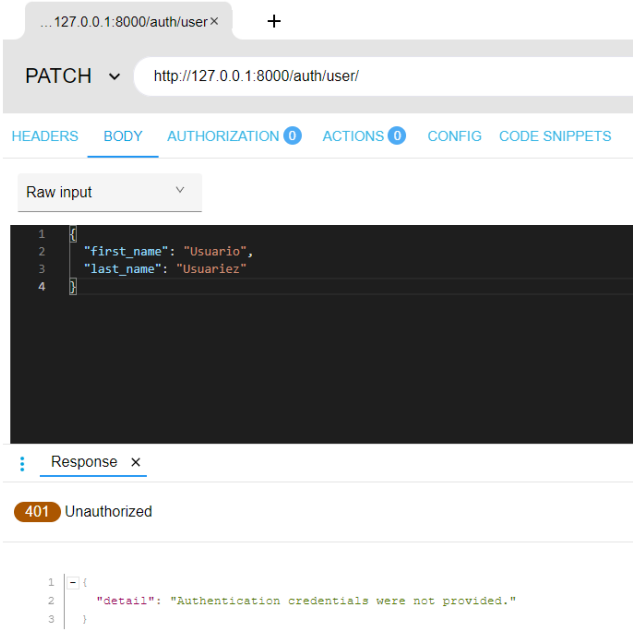
Hasta ahora, hemos visto todo el funcionamiento interno, lo que pueden ver los administradores en el panel de administrador, y lo que verán los desarrolladores *front-end* en el *Swagger*. Pero, ¿cómo funciona con un usuario normal realizando las peticiones? Para ver esto, vamos a simular con Advanced REST Client las peticiones que realizaría un usuario a la hora de usar la *API*.



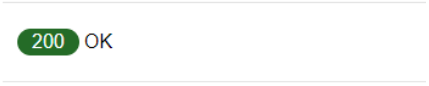
Para ello, el usuario realizará una petición *POST* al *endpoint* de registro. La petición tendrá un header que indicará que el contenido que envía es json.

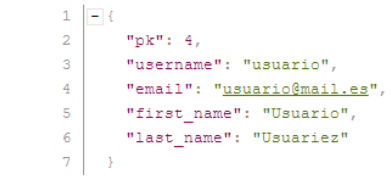


Enviando el diccionario de datos con todo correcto, la *API* devuelve un *status code 201*, mandando un *token* de acceso, un *token* de refresco y la información serializada del usuario recién creado.

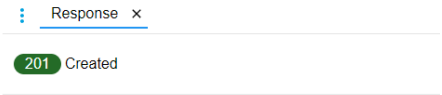


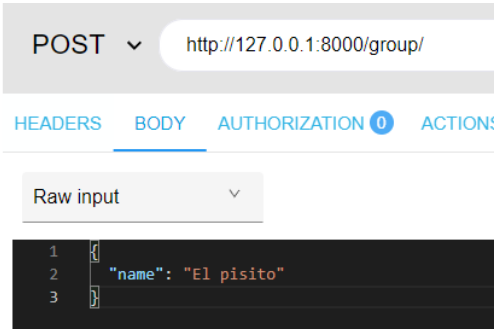
Si intentamos ahora realizar una petición, nos dará este error, ya que necesitamos añadir el *header* con el *token* de autorización a las peticiones que realicemos, poniendo detrás de la palabra "*Bearer*" y un espacio.





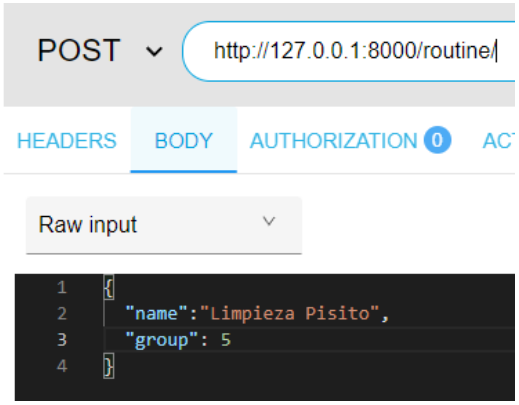
De esta manera, se puede modificar parcialmente los datos del usuario.





Petición

Respuesta



```
{
  "routine": 5,
  "title": "Limpiar cocina",
  "starts_at": "2023-06-23T19:00:00",
  "assigned_to": 4
}
```

Asigno la tarea a él mismo, ya que es el único usuario en el grupo, de otra forma, saltaría un error 400.

201 Created


```
{
  "id": 4,
  "routine": 5,
  "title": "Limpiar cocina",
  "description": null,
  "starts_at": "2023-06-23T19:00:00Z",
  "created_by": {
    "id": 4,
    "username": "usuario",
    "first_name": "Usuario",
    "last_name": "Usuariez",
    "email": "usuario@mail.es"
  },
  "assigned_to": {
    "id": 4,
    "username": "usuario",
    "first_name": "Usuario",
    "last_name": "Usuariez",
    "email": "usuario@mail.es"
  },
  "completed": false,
  "created_at": "2023-06-19T23:58:54.686405Z",
  "updated_at": "2023-06-19T23:58:54.686405Z"
}
```


Ahora vamos a hacer un listado de Groups (devolverá únicamente 1, ya que este usuario solamente pertenece al grupo que acaba de crear)


```
{
  "lastPage": 1,
  "currentPage": 1,
  "perPage": 10,
  "total": 1,
  "data": [
    {
      "id": 5,
      "name": "El pisito",
      "user_owner": {
        "id": 4,
        "username": "usuario",
        "first_name": "Usuario",
        "last_name": "Usuariez",
        "email": "usuario@mail.es"
      },
      "users_list": [
        {
          "id": 4,
          "username": "usuario",
          "first_name": "Usuario",
          "last_name": "Usuariez",
          "email": "usuario@mail.es"
        }
      ],
      "routines": [
        {
          "id": 5,
          "name": "Limpieza Pisito",
          "group": 5,
          "tasks": [
            {
              "id": 4,
              "routine": 5,
              "title": "Limpiar cocina",
              "description": null,
              "starts_at": "2023-06-23T19:00:00Z",
              "created_by": {
                "id": 4,
                "username": "usuario",
                "first_name": "Usuario",
                "last_name": "Usuariez",
                "email": "usuario@mail.es"
              },
              "assigned_to": {
                "id": 4,
                "username": "usuario",
                "first_name": "Usuario",
                "last_name": "Usuariez",
                "email": "usuario@mail.es"
              },
              "completed": false,
              "created_at": "2023-06-19T23:58:54.686405Z",
              "updated_at": "2023-06-19T23:58:54.686405Z"
            }
          ],
          "created_at": "2023-06-19T23:54:44.129136Z",
          "updated_at": "2023-06-19T23:54:44.129136Z"
        }
      ],
      "created_at": "2023-06-19T23:51:00.481566Z",
      "updated_at": "2023-06-19T23:51:00.481566Z"
    }
  ]
}
```

Ya que el serializador de *Group* incorpora un listado de *Routines* serializadas y a su vez, el serializador de *Routine* incorpora un serializador de un listado de *Tasks*, podemos obtener toda la información de los grupos a los cuales pertenece un usuario (en este caso, únicamente a uno), con una única petición, en caso de hacerla en el *endpoint* de su(s) grupo(s).

7. Manual de instalación/despliegue

Para esto necesitamos un intérprete de *Python3.10* o superior. Además, tendremos que tener *git* en nuestro equipo. Por último, necesitamos tener instalado *pgAdmin* y *postgreSQLServer* en nuestro equipo.

El primer paso será clonar mi repositorio (https://github.com/raulfc2000/home_task_tracker_api)

Ahora tendrás que crear tu entorno virtual, con el comando "*python3 -m venv [nombre_de_tu_entorno]*". Recomiendo poner algo como *.env*, o *venv*.

Ahora, tendrás que instalar las librerías, utilizando el comando "*pip install -r requirements.txt*". Es posible que sea necesario *upgradear* el *pip*. Si es necesario, esto se puede hacer con el comando "*python -m pip install --upgrade pip*" (en *Windows*).

Para ahora poder sincronizar la base de datos con la *API*, usaremos *pgAdmin*, en un servidor cualquiera, crea una base de datos con las credenciales del fichero *settings.py*.

Por último, lanza el comando "*python.manage.py migrate*", para sincronizar las migraciones del proyecto con la base de datos, y luego el comando "*python manage.py runserver*". Con esto, ¡habrás lanzado el servidor en local, en el puerto 8000!

4. DIFICULTADES ENCONTRADAS EN LA REALIZACIÓN DEL PROYECTO

A pesar de haber trabajado con esta tecnología en equipo en la empresa de prácticas, definitivamente el organizar todo por mi cuenta, tener en cuenta todas las casuísticas, diría que mi mayor problema ha sido la organización a nivel de arquitectura de datos. Al final estoy bastante contento con el resultado, ya que he tenido que hacer muchos cambios por errores que me iban saliendo y teniendo que borrar la base de datos y empezando de nuevo por inconcluencias que no he visto hasta que ha fallado. Creo que a nivel organizativo he aprendido bastante en ese aspecto.

Por otro lado, algo realmente difícil para mí, a pesar de que sean pocas líneas de código, ha sido el modificar el *queryset* en las *views*, porque no sabía muy bien como gestionar los permisos, y tuve muchos fallos de que el usuario pudiera ver grupos a los que no pertenecía y cosas similares.

5. PROPUESTAS DE MEJORA

Ha habido varias cosas que no he podido implementar, algunas que imaginaba que no me daría tiempo, y otras que sí, realmente me han frustrado.

Lo que más me ha frustrado ha sido el testeo. Considero que una de las cosas que más he aprendido en las prácticas es el testeo unitario para los filtros de la view, testeo de serializador, de integración, mockeo… Por eso, era algo que quería implementar sí o sí a esta entrega del proyecto. Desafortunadamente, no he avanzado tan rápido como me hubiera gustado y no me ha dado tiempo a añadirlo. Mi idea era usar las librerías de *django.test*, *pytest*, *model_mommy* y la de *unittest.mock*, para testear todo el proyecto, ni siquiera voy a hablar de ellas porque no he podido usarlas.

Las dos cosas que, como objetivo personal, eran opcionales, pero me hubiera gustado añadir, era, por un lado, el servidor de correo para aumentar la seguridad, haciendo disponibles los *endpoints* que mencioné en su apartado y por otro lado me hubiera gustado añadirle autenticación social con cuenta de *Google*.

6. CONCLUSIÓN FINAL

Este proyecto me ha servido para entender lo que es encargarme del 100% de algo un poco más ambicioso, desde la planificación hasta todo el desarrollo. Creo que ahora entiendo mucho mejor las bases del *framework* y creo que el hacer toda la *API* yo, me ha ayudado también a entender mucho mejor el funcionamiento más interno. Sin duda, tengo muchísimo que aprender, pero este proyecto para mí, es como el primer paso que doy como "pollito recién salido del cascarón".

Agradecimientos por ver este, mi proyecto, al que le he dedicado mucho tiempo y esfuerzo.

7. BIBLIOGRAFÍA

Allauth: https://django-allauth.readthedocs.io/en/latest/

Django REST Framework: https://www.django-rest-framework.org/

django-filter: https://django-filter.readthedocs.io/en/stable/

dj-rest-auth: https://dj-rest-auth.readthedocs.io/en/latest/

Markdown: https://markdown.es/

pgAdmin: https://www.pgadmin.org/

PostgreSQL: https://www.postgresql.org/

Python: https://www.python.org/

SimpleJWT: https://django-rest-framework-simplejwt.readthedocs.io/en/latest/

Swagger: https://django-rest-swagger.readthedocs.io/en/latest/

Agradecimientos al instituto IES Belén y su equipo docente, y a Grupo Deidev, por la experiencia y por los conocimientos adquiridos.