EBook Gratis

APRENDIZAJE

django-rest-framework

Free unaffiliated eBook created from **Stack Overflow contributors.**

#django-rest-framework

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django-rest-framework

It is an unofficial and free django-rest-framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official django-rest-framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con django-rest-framework

Observaciones

Django REST Framework es un conjunto de herramientas para crear aplicaciones web. Ayuda al programador a realizar *API REST* , pero puede manejar niveles de API menos maduros. Para obtener más información sobre los niveles de madurez de la API, busque el modelo de madurez de Richardson.

En particular, Django REST Framework no respalda ningún diseño de nivel de Hipermedia en particular, y depende del programador (u otros proyectos, como srf-hal-json) si desean seguir una implementación de la API de HATEOAS, establecer su Opiniones fuera del marco. Por lo tanto, es posible implementar una API HATEOAS en el marco REST de Django, pero ya no hay utilidades disponibles.

Versiones

Versión	Fecha de lanzamiento
3.5.3	2016-11-07

Examples

Instalar o configurar

Requerimientos

- Python (2.7, 3.2, 3.3, 3.4, 3.5, 3.6)
- Django (1.7+, 1.8, 1.9, 1.10, 1.11)

Instalar

Puede usar `pip` para instalar o clonar el proyecto desde github.

- **Usando `pip`**

```
pip install djangorestframework
```

- **Usando `git clone` :**

```
git clone git@github.com:tomchristie/django-rest-framework.git
```

Después de la instalación, debe **agregar `rest_framework` a la configuración de `INSTALLED_APPS` .**

```
INSTALLED_APPS = (
    ...
    'rest_framework',
)
```

Si pretende utilizar la API navegable, probablemente también desee agregar las vistas de inicio y cierre de sesión del marco REST. Agregue lo siguiente a su raíz urls.py archivo.

```
urlpatterns = [
    ...
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Ejemplo

Veamos un ejemplo rápido del uso del marco REST para crear una API simple respaldada por modelos.

Crearemos una API de lectura y escritura para acceder a la información de los usuarios de nuestro proyecto.

Cualquier configuración global para una API de estructura REST se mantiene en un único diccionario de configuración denominado `REST_FRAMEWORK` . Comience agregando lo siguiente a su módulo `settings.py` :

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Estamos listos para crear nuestra API ahora. Aquí está nuestro módulo de `urls.py` raíz de nuestro proyecto:

```
from django.conf.urls import url, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'is_staff')

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

```
# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Ahora puede abrir la API en su navegador en `http://127.0.0.1:8000/` , y ver su nueva API de 'usuarios'. Si usa el control de inicio de sesión en la esquina superior derecha, también podrá agregar, crear y eliminar usuarios del sistema.

Lea Empezando con django-rest-framework en línea: https://riptutorial.com/es/django-rest-framework/topic/1875/empezando-con-django-rest-framework

Capítulo 2: Autenticación

Examples

Escritura de autenticaciones personalizadas

desde django.contrib.auth.models importar Usuario desde rest_framework importar autenticación desde rest_framework excepciones de importación

Esta autenticación de ejemplo es directamente de los documentos oficiales aquí .

```
class ExampleAuthentication(BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('X_USERNAME')
        if not username:
            return None

        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('No such user')

        return (user, None)
```

Hay cuatro partes en una clase de autenticación personalizada.

1. Amplíelo desde la clase BaseAuthentication proporcionada en `from rest_framework. autenticación de base de autenticación de autenticación
2. Tener un método llamado `authenticate request` toma como primer argumento.
3. Devuelva una tupla de (usuario, Ninguna) para una autenticación exitosa.
4. Aumente la excepción `AuthenticationFailed` para una autenticación fallida. Esto está disponible en rest_framework.authentication.

```
class SecretAuthentication(BaseAuthentication):
    def authenticate(self, request):
        app_key = request.META.get('APP_KEY')
        app_secret = request.META.get('APP_SECRET')
        username = request.META.get('X_USERNAME')
        try:
            app = ClientApp.objects.match_secret(app_key, app_secret)
        except ClientApp.DoesNotExist:
            raise AuthenticationFailed('App secret and key does not match')
        try:
           user = app.users.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('Username not found, for the specified app')
        return (user, None)
```

El esquema de autenticación devolverá HTTP 403 Respuestas prohibidas cuando se deniegue el acceso a una solicitud no autenticada.

Autenticación de solicitud base API-KEY

Puede usar las clases de permisos de `django rest framework` para verificar los encabezados de las solicitudes y autenticar las solicitudes de los usuarios

- Define tu secret_key en la configuración del proyecto

```
API_KEY_SECRET = 'secret_value'
```

nota: una buena práctica es utilizar variables de entorno para almacenar este valor secreto.

- Definir una clase de permiso para la autenticación API-KEY.

cree el archivo permissions.py en el directorio de su aplicación con los siguientes códigos:

```
from django.conf import settings

from rest_framework.permissions import BasePermission

class Check_API_KEY_Auth(BasePermission):
    def has_permission(self, request, view):
        # API_KEY should be in request headers to authenticate requests
        api_key_secret = request.META.get('API_KEY')
        return api_key_secret == settings.API_KEY_SECRET
```

- Añadir esta clase de permiso a las vistas

```
from rest_framework.response import Response
from rest_framework.views import APIView

from .permissions import Check_API_KEY_Auth


class ExampleView(APIView):
    permission_classes = (Check_API_KEY_Auth,)

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

O, si está utilizando el decorador @api_view con vistas basadas en funciones

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.response import Response

from .permissions import Check_API_KEY_Auth


@api_view(['GET'])
@permission_classes((Check_API_KEY_Auth, ))
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
```

```
    return Response(content)
```

Lea Autenticación en línea: https://riptutorial.com/es/django-rest-framework/topic/5451/autenticacion

Capítulo 3: Autenticación

Examples

Configurando la autenticación globalmente

La autenticación en Django REST Framework se puede configurar globalmente como una subclave de la variable `REST_FRAMEWORK` en settings.py, al igual que el resto de las configuraciones de framework predeterminadas.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

Configuración de la autenticación para un APIView específico

La autenticación se puede configurar para un punto final APIView específico, utilizando la variable `authentication_classes` :

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)

    def get(self, request):
        content = {
            'user': unicode(request.user),  # `django.contrib.auth.User` instance.
            'auth': unicode(request.auth),  # None
        }
        return Response(content)
```

Usando autenticación básica basada en token

Django REST Framework proporciona un mecanismo de autenticación básico basado en token que debe configurarse como una aplicación en Django antes de poder ser utilizado, de modo que los tokens se creen en la base de datos y se maneje su ciclo de vida.

Añadir autenticación basada en token a settings.py

```
INSTALLED_APPS = (
    ...
    'rest_framework.authtoken'
)
```

Ejecutar la migración de la base de datos.

```
./manage.py migrate
```

Crea tokens para tus usuarios

De alguna manera, tendrás que crear un token y devolverlo:

```
def some_api(request):
    token = Token.objects.create(user=request.user)
    return Response({'token': token.key})
```

Ya hay un punto final de API en la aplicación del token, por lo que simplemente puede agregar lo siguiente a su urls.py:

urls.py

```
from rest_framework.authtoken import views
urlpatterns += [
    url(r'^auth-token/', views.obtain_auth_token)
]
```

Los clientes ahora pueden autenticarse

Usando un encabezado de `Authorization` como:

```
Authorization: Token 123456789
```

Prefijado por un "token" literal y el token en sí después del espacio en blanco.

El literal se puede cambiar subclasificando `TokenAuthentication` y cambiando la variable de clase de `keyword` .

Si se autentica, `request.auth` contendrá la instancia `rest_framework.authtoken.models.BasicToken` .

Configuración de la autenticación OAuth2

OAuth no se maneja con Django REST Framework, pero hay un par de módulos pip que implementan un cliente OAuth. La documentación del marco REST sugiere uno de los siguientes

módulos:

- Django OAuth Toolkit
- Django REST Framework OAuth

Django OAuth Toolkit

```
pip install django-oauth-toolkit
```

settings.py

```
INSTALLED_APPS = (
    ...
    'oauth2_provider',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'oauth2_provider.ext.rest_framework.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
)
```

Django REST Framework OAuth

```
pip install djangorestframework-oauth django-oauth2-provider
```

settings.py

```
INSTALLED_APPS = (
    ...
    'provider',
    'provider.oauth2',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^oauth2/', include('provider.oauth2.urls', namespace='oauth2')),
)
```

Administración

Vaya al panel de administración y cree un nuevo `Provider.Client` para tener un `client_id` y un `client_secret` .

Usar una mejor autorización basada en token con varios tokens de cliente

El paquete más interesante para administrar tokens reales es django-rest-knox, que admite múltiples tokens por usuario (y cancela cada token de forma independiente), además de contar con soporte para la caducidad del token y varios otros mecanismos de seguridad.

`django-rest-knox` depende de la `cryptography` . Puede encontrar más información sobre cómo instalarlo en: http://james1345.github.io/django-rest-knox/installation/

Instalando knox

```
pip install django-rest-knox
```

settings.py

```
INSTALLED_APPS = (
  ...
  'rest_framework',
  'knox',
  ...
)
```

Aplicar migraciones:

```
./manage.py migrate
```

Lea Autenticación en línea: https://riptutorial.com/es/django-rest-framework/topic/6276/autenticacion

Capítulo 4: Autenticación de tokens con Django Rest Framework

Examples

AÑADIR FUNCIONALIDAD AUTOMÁTICA

Django REST Framework ya tiene algunos métodos de autenticación incorporados, uno de ellos está basado en Token, así que lo primero que debe hacer es decirle a nuestro proyecto que vamos a utilizar la autenticación del resto de marcos. Abra el archivo `settings.py` y agregue la línea resaltada.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'test_app',
)
```

Como hemos agregado una nueva aplicación al proyecto, debemos sincronizar

```
python manage.py migrate
```

CREANDO UN SUPERUSUARIO EN DJANGO

Para utilizar la autenticación, podemos confiar en el modelo de usuarios de django, así que lo primero que debemos hacer es crear un superusuario.

```
python manage.py createsuperuser
```

OBTENER EL USUARIO TOKEN

La funcionalidad de autenticación del token asigna un token a un usuario, por lo que cada vez que use ese token, el objeto de solicitud tendrá un atributo de usuario que contiene la información del modelo del usuario. Fácil, ¿no es así?

Crearemos un nuevo método POST para devolver el token para este usuario, siempre que la solicitud contenga un usuario y una contraseña correctos. Abra **views.py** ubicado en la carpeta de aplicación test_app.

```
from rest_framework.response import Response
```

```
from rest_framework.authtoken.models import Token
from rest_framework.exceptions import ParseError
from rest_framework import status

from django.contrib.auth.models import User


# Create your views here.
class TestView(APIView):
    """
    """

    def get(self, request, format=None):
        return Response({'detail': "GET Response"})

    def post(self, request, format=None):
        try:
            data = request.DATA
        except ParseError as error:
            return Response(
                'Invalid JSON - {0}'.format(error.detail),
                status=status.HTTP_400_BAD_REQUEST
            )
        if "user" not in data or "password" not in data:
            return Response(
                'Wrong credentials',
                status=status.HTTP_401_UNAUTHORIZED
            )

        user = User.objects.first()
        if not user:
            return Response(
                'No default user, please create one',
                status=status.HTTP_404_NOT_FOUND
            )

        token = Token.objects.get_or_create(user=user)

        return Response({'detail': 'POST answer', 'token': token[0].key})
```

USANDO EL TOKEN

Vamos a crear una nueva vista que requiera este mecanismo de autenticación.

Necesitamos agregar estas líneas de importación:

```
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
```

y luego crea la nueva Vista en el mismo archivo `views.py`

```
class AuthView(APIView):
    """
    Authentication is needed for this methods
    """
    authentication_classes = (TokenAuthentication,)
```

```
    permission_classes = (IsAuthenticated,)

    def get(self, request, format=None):
        return Response({'detail': "I suppose you are authenticated"})
```

Como hicimos en la publicación anterior, debemos informarle a nuestro proyecto que tenemos una nueva ruta REST en la escucha, en `test_app/urls.py`

```
from rest_framework.urlpatterns import format_suffix_patterns
from test_app import views

urlpatterns = patterns('test_app.views',
    url(r'^', views.TestView.as_view(), name='test-view'),
    url(r'^auth/', views.AuthView.as_view(), name='auth-view'),
)

urlpatterns = format_suffix_patterns(urlpatterns)
```

Utilizando CURL

Si un rizo se ejecuta en este punto final

```
curl http://localhost:8000/auth/
op : {"detail": "Authentication credentials were not provided."}%
```

devolvería un **error 401 NO AUTORIZADO**

pero en caso de que tengamos un token antes:

```
curl -X POST -d "user=Pepe&password=aaaa"  http://localhost:8000/
{"token": "f7d6d027025c828b65cee5d38240aec60dffa150", "detail": "POST answer"}%
```

y luego ponemos ese token en el encabezado de la solicitud de esta manera:

```
curl http://localhost:8000/auth/ -H 'Authorization: Token
f7d6d027025c828b65cee5d38240aec60dffa150'

op: {"detail": "I suppose you are authenticated"}%
```

Lea Autenticación de tokens con Django Rest Framework en línea: https://riptutorial.com/es/django-rest-framework/topic/9087/autenticacion-de-tokens-con-django-rest-framework

Capítulo 5: Enrutadores

Introducción

El enrutamiento es el proceso de mapear la lógica (ver métodos, etc.) a un conjunto de URL. El marco REST agrega soporte para enrutamiento de URL automático a Django.

Sintaxis

- enrutador = enrutadores.SimpleRouter ()
- router.register (prefijo, conjunto de vistas)
- router.urls # el conjunto generado de urls para el conjunto de vistas registrado.

Examples

[Introductorio] Uso básico, SimpleRouter

Enrutamiento automático para el DRF, se puede lograr para las clases `ViewSet` .

1. Supongamos que la clase ViewSet va por el nombre de `MyViewSet` para este ejemplo.

2. Para generar el enrutamiento de `MyViewSet` , se utilizará el `SimpleRouter` .
   En `myapp/urls.py` :

   ```
   from rest_framework import routers

   router = routers.SimpleRouter()         # initialize the router.
   router.register(r'myview', MyViewSet)   # register MyViewSet to the router.
   ```

3. Eso generará los siguientes patrones de URL para `MyViewSet` :

   - `^myview/$` con el nombre `myview-list` .
   - `^myview/{pk}/$` con el nombre `myview-detail`

4. Finalmente, para agregar los patrones generados en los patrones de URL de `myapp` , se usará el `include()` django.
   En `myapp/urls.py` :

   ```
   from django.conf.urls import url, include
   from rest_framework import routers

   router = routers.SimpleRouter()
   router.register(r'myview', MyViewSet)

   urlpatterns = [
       url(r'other/prefix/if/needed/', include(router.urls)),
   ]
   ```

Lea Enrutadores en línea: https://riptutorial.com/es/django-rest-framework/topic/10938/enrutadores

Capítulo 6: Filtros

Examples

Ejemplos de filtrado, desde los simples a los más complejos

Filtrado simple de vainilla

Para filtrar cualquier vista, anule su método `get_queryset` para devolver un conjunto de consultas filtradas

```
class HREmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

Todas las funciones de la API utilizarán el conjunto de consultas filtradas para las operaciones. Por ejemplo, la lista de la vista anterior solo contendrá empleados cuyo departamento es Recursos Humanos.

Accediendo a los parámetros de consulta en get_queryset

El filtrado basado en los parámetros de solicitud es fácil.

`self.request` , `self.args` y `self.kwargs` están disponibles y apuntan a la solicitud actual y sus parámetros para el filtrado

```
def DepartmentEmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department=self.kwargs["department"])
```

Dejar que los parámetros de la API decidan qué filtrar

Si desea más flexibilidad y permite que la llamada a la API pase parámetros para filtrar la vista, puede agregar filtros de backends como Django Request Framework (instalado a través de pip)

```
from rest_framework import filters

class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
```

```
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)
```

Ahora puede hacer una llamada a `/api/employees?department=Human Resources` y obtendrá una lista de empleados que pertenecen solo al departamento de Recursos Humanos, o `/api/employees?role=manager&department=Human Resources` para que solo entren los gerentes El departamento de recursos humanos.

Puede combinar el filtrado de conjuntos de consultas con Django Filter Backend, sin ningún problema. Los filtros funcionarán en el conjunto de consultas filtradas devuelto por `get_queryset`

```
from rest_framework import filters

class HREmployees(generics.ListAPIView):
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)

    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

FilterSets

Hasta ahora, usted puede arreglárselas con coincidencias de tipo simple en los casos anteriores.

Pero, ¿qué sucede si desea algo más complejo, como una lista de empleados de recursos humanos que tienen entre 25 y 32 años de edad?

Respuesta al problema: Conjuntos de filtros

Los conjuntos de filtros son clases que definen cómo filtrar varios campos del modelo.

Defínelos como tal

```
class EmployeeFilter(django_filters.rest_framework.FilterSet):
    min_age = filters.django_filters.NumberFilter(name="age", lookup_expr='gte')
    max_age = filters.django_filters.NumberFilter(name="price", lookup_expr='lte')

    class Meta:
        model = Employee
        fields = ['age', 'department']
```

`name` apunta al campo que desea filtrar

`lookup_expr` básicamente se refiere a los mismos nombres que usa al filtrar los conjuntos de consultas, por ejemplo, puede hacer una coincidencia "comienza con" usando `lookup_expr="startswith"` que es equivalente a `Employee.objects.filter(department__startswith="Human")`

Luego `filter_class` en sus clases de vista usando `filter_class` lugar de `filter_fields`

```
class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_class = EmployeeFilter
```

Ahora puede hacer `/api/employees?department=Human Resources&min_age=25&max_age=32`

Coincidencias y relaciones no exactas

Las clases y expresiones de filtro son muy similares a cómo se especifica el filtrado en los conjuntos de consultas

Puede usar la notación "__" para filtrar campos en las relaciones. Por ejemplo, si el departamento era una clave externa de un empleado, puede agregar

```
filter_fields=("department__name",)
```

y luego puedes hacer `/api/employees?department__name=Human Resources`

O más elegante, puede crear un conjunto de filtros, agregar una variable de filtro llamada `dept` y establecer su nombre en `department__name` , lo que le permite hacer `/api/employees?dept=Human Resources`

Lea Filtros en línea: https://riptutorial.com/es/django-rest-framework/topic/8144/filtros

Capítulo 7: Mixins

Introducción

Las clases de mezcla proporcionan las acciones que se utilizan para proporcionar el comportamiento de vista básico. Tenga en cuenta que las clases mixtas proporcionan métodos de acción en lugar de definir los métodos del controlador, como `.get()` y `.post()` , directamente. Esto permite una composición más flexible del comportamiento. - Documentación oficial del marco de Django Rest -

Examples

[Introductoria] Lista de mixins y uso en vistas / conjuntos de vistas

---

Lista de mixins disponibles:

- **ListModelMixin:** proporciona un método `.list()` a la vista / conjunto de vistas
- **RetrieveModelMixin:** proporciona un método `.retrieve()` a la vista / conjunto de vistas
- **CreateModelMixin:** proporciona un método `.create()` a la vista / conjunto de vistas
- **UpdateModelMixin:** proporciona un método `.update()` a la vista / conjunto de vistas
- **DestroyModelMixin:** proporciona un método `.destroy()` a la vista / conjunto de vistas

---

Podemos mezclar y combinar los mixins en nuestras vistas genéricas y conjuntos de vistas, para darles la utilidad correspondiente que necesitamos:

1. ¿Una vista de API con los `.list()` `.create()` y `.destroy()` ?
   Heredar de `GenericAPIView` y combinar los mixins apropiados:

```
from rest_framework import mixins, generics

from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyCustomAPIView(mixins.ListModelMixin,
                      mixins.CreateModelMixin,
                      mixins.DestroyModelMixin,
                      generics.GenericAPIView):

    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
```

```
        return self.destroy(request, *args, **kwargs)
```

2. ¿Un conjunto de vistas con solo los `.list()` y `.update()` ? `GenericViewSet` y agregue los mixins apropiados:

```
from rest_framework import mixins

class MyCustomViewSet(mixins.ListModelMixin,
                      mixins.UpdateModelMixin,
                      viewsets.GenericViewSet):
    pass
```

   Sí, fue así de fácil !!

Para concluir, podemos combinar y combinar cada mezcla que necesitemos y utilizarla para personalizar nuestras vistas y sus métodos en cualquier combinación posible.

[Intermedio] Creando Mixins personalizados

DRF ofrece la oportunidad de personalizar aún más el comportamiento de las vistas / conjuntos de vistas genéricos al permitir la creación de mixins personalizados.

Cómo:

Para definir una mezcla personalizada solo necesitamos crear una clase heredada de un `object` .

Supongamos que queremos definir dos vistas separadas para un modelo llamado `MyModel` . Esas vistas compartirán el mismo conjunto de `queryset` y la misma `queryset serializer_class` . Nos ahorraremos algo de repetición de código y pondremos lo anterior en una sola mezcla para ser heredada por nuestros puntos de vista:

- `my_app/views.py` (esa no es la única opción de archivo disponible para colocar nuestros mixins personalizados, pero es la menos compleja):

```
from rest_framework.generics import CreateAPIView, RetrieveUpdateAPIView
from rest_framework.permissions import IsAdminUser

class MyCustomMixin(object):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

class MyModelCreateView(MyCustomMixin, CreateAPIView):
    """
    Only an Admin can create a new MyModel object
    """
    permission_classes = (IsAdminUser,)

    Do view staff if needed...

class MyModelCreateView(MyCustomMixin, RetrieveUpdateAPIView):
    """
```

```
    Any user can Retrieve and Update a MyModel object
    """
    Do view staff here...
```

Conclusión:

Los mixins son esencialmente bloques de código reutilizable para nuestra aplicación.

Lea Mixins en línea: https://riptutorial.com/es/django-rest-framework/topic/10083/mixins

Capítulo 8: Paginación

Introducción

La paginación es la división de grandes conjuntos de datos en páginas separadas y autónomas.

En django rest framework, la paginación le permite al usuario modificar la cantidad de datos en cada página y el estilo por el cual se aplica la división.

Examples

[Introductorio] Configuración del estilo de paginación globalmente

Para configurar el estilo de paginación para todo el proyecto, debe establecer `DEFAULT_PAGINATION_CLASS` y `PAGE_SIZE` en la configuración del proyecto.

Para hacerlo, vaya a `settings.py` y en la variable `REST_FRAMEWORK` , agregue lo siguiente:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.DESIRED_PAGINATION_STYLE',
    'PAGE_SIZE': 100
}
```

En lugar de `DESIRED_PAGINATION_STYLE` se debe colocar uno de los siguientes:

- `PageNumberPagination` : Acepta un número de `page` único en los parámetros de consulta de solicitud.

  ```
  http://your_api_url/a_table/?page=2
  ```

- `LimitOffsetPagination` : Acepta un parámetro de `limit` , que indica el número máximo de elementos que se devolverán y un parámetro de `offset` que indica la posición inicial de la consulta en relación con el conjunto de datos. `PAGE_SIZE` no necesita configurarse para este estilo.

  ```
  http://your_api_url/a_table/?limit=50&offset=100
  ```

- `CursorPagination` : la paginación basada en el cursor es más compleja que los estilos anteriores. Requiere que el conjunto de datos presente un ordenamiento fijo, y no permite que el cliente navegue hacia posiciones arbitrarias del conjunto de datos.

- Se pueden definir estilos de paginación personalizados en lugar de los anteriores.

[Intermedio] Anule el estilo de Paginación y configure la Paginación por clase

Anular el estilo de paginación:

Cada estilo de paginación disponible puede ser anulado creando una nueva clase que herede de uno de los estilos disponibles y luego altera sus parámetros:

```
class MyPagination(PageNumberPagination):
    page_size = 20
    page_size_query_param = 'page_size'
    max_page_size = 200
    last_page_strings = ('the_end',)
```

Esos parámetros (tal como figuran en la documentación oficial de la paginación) son:

PageNumberPagination

- `page_size` : un valor numérico que indica el tamaño de la página. Si se establece, esto anula la configuración de `PAGE_SIZE` . El valor predeterminado es el mismo que el de la clave de configuración `PAGE_SIZE` .
- `page_query_param` : un valor de cadena que indica el nombre del parámetro de consulta que se usará para el control de paginación.
- `page_size_query_param` : Si se establece, este es un valor de cadena que indica el nombre de un parámetro de consulta que le permite al cliente establecer el tamaño de la página por solicitud. El valor predeterminado es `None` , lo que indica que el cliente no puede controlar el tamaño de página solicitado.
- `max_page_size` : si se establece, este es un valor numérico que indica el tamaño de página solicitado máximo permitido. Este atributo solo es válido si `page_size_query_param` también está configurado.
- `last_page_strings` : una lista o tupla de valores de cadena que indican los valores que se pueden usar con `page_query_param` para solicitar la página final del conjunto. El valor predeterminado es `('last',)`
- `template` : el nombre de una plantilla para usar al representar controles de paginación en la API navegable. Puede anularse para modificar el estilo de representación o establecerlo en `None` para deshabilitar completamente los controles de paginación HTML. El valor predeterminado es `"rest_framework/pagination/numbers.html"` .

LimitOffsetPagination

- `default_limit` : un valor numérico que indica el límite a usar si el cliente no proporciona uno en un parámetro de consulta. El valor predeterminado es el mismo que el de la clave de configuración `PAGE_SIZE` .
- `limit_query_param` : un valor de cadena que indica el nombre del parámetro de consulta "límite". El valor predeterminado es `'limit'` .
- `offset_query_param` : un valor de cadena que indica el nombre del parámetro de consulta "offset". El valor predeterminado es `'offset'` .
- `max_limit` : si se establece, este es un valor numérico que indica el límite máximo permitido que puede solicitar el cliente. Por defecto, `None` .
- `template` : igual que *PageNumberPagination* .

Cursor cursor

- `page_size` : igual que *PageNumberPagination* .
- `cursor_query_param` : un valor de cadena que indica el nombre del parámetro de consulta "cursor". El valor predeterminado es `'cursor'` .
- `ordering` : debe ser una cadena o una lista de cadenas, que indica el campo en el que se aplicará la paginación basada en el cursor. Por ejemplo: `ordering = 'slug'` . El valor predeterminado es `-created` . Este valor también puede anularse utilizando `OrderingFilter` en la vista.
- `template` : igual que *PageNumberPagination* .

Configuración de la paginación por clase:

Además de la capacidad de configurar el estilo de Paginación globalmente, está disponible una configuración por clase:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = LimitOffsetPagination
```

Ahora solo `MyViewSet` tiene una paginación `LimitOffsetPagination` .

Un estilo de paginación personalizado se puede utilizar de la misma manera:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = MyPagination
```

[Intermedio] Ejemplo de uso complejo

Supongamos que tenemos una API compleja, con muchas vistas genéricas y algunos conjuntos de vistas genéricos. Queremos habilitar `PageNumberPagination` para cada vista, excepto una (ya sea vista o conjunto de vistas genérico, no hace una diferencia) para lo que queremos un caso personalizado de `LimitOffsetPagination` .

Para lograrlo necesitamos:

1. En `settings.py` colocaremos nuestra paginación predeterminada para habilitarla para cada vista / conjunto de vistas genérico y estableceremos `PAGE_SIZE` en 50 elementos:

   ```
   REST_FRAMEWORK = {
       'DEFAULT_PAGINATION_CLASS':
           'rest_framework.pagination.PageNumberPagination',
       'PAGE_SIZE': 50
   }
   ```

2. Ahora en nuestro `views.py` (o en otro `.py` ex: `paginations.py`), necesitamos anular el `LimitOffsetPagination` :

   ```
   from rest_framework.pagination import LimitOffsetPagination

   class MyOffsetPagination(LimitOffsetPagination):
       default_limit = 20
       max_limit = 1000
   ```

Una `LimitOffsetPagination` personalizada con `PAGE_ZISE` de 20 artículos y un `limit` máximo de 1000 artículos.

3. En nuestro `views.py` , necesitamos definir la `pagination_class` de nuestra vista especial:

```
imports ...

# ====================================
#    PageNumberPagination classes
# ====================================

class FirstView(generics.ListAPIView):
    ...

class FirstViewSet(viewsets.GenericViewSet):
    ...

...

# ====================================
#     Our custom Pagination class
# ====================================

class IAmSpecialView(generics.ListAPIView):
    pagination_class = MyOffsetPagination
    ...
```

Ahora, cada vista / conjunto de vistas genérico en la aplicación tiene `PageNumberPagination` , excepto la clase `IAmSpecial` , que es realmente *especial* y tiene su propia `LimitOffsetPagination` personalizada.

[Avanzado] Paginación sobre vistas / conjuntos de vistas no genéricos

Este es un tema avanzado, no intente sin entender primero los otros ejemplos de esta página .

Como se indica en el marco oficial de Django Rest en paginación :

> La paginación solo se realiza automáticamente si está utilizando las vistas genéricas o los conjuntos de vistas. Si está utilizando una APIView normal, deberá llamar a la API de paginación para asegurarse de que devuelve una respuesta paginada. Consulte el código fuente de las clases mixins.ListModelMixin y generics.GenericAPIView para ver un ejemplo.

¿Pero qué sucede si queremos utilizar la paginación en una vista / conjunto de vistas no genérico?

Bueno, vamos a bajar el agujero del conejo:

1. La primera parada es el repositorio oficial de Django Rest Framework y específicamente el django-rest-framework / rest_framework / generics.py . La línea específica a la que apunta este enlace, nos muestra cómo los desarrolladores del marco tratan con la paginación en sus genéricos.
   ¡Eso es exactamente lo que vamos a utilizar para nuestra vista también!

2. Supongamos que tenemos una configuración de paginación global como la que se muestra en el ejemplo introductorio de esta página y supongamos también que tenemos una vista `APIView` que queremos aplicar la paginación.

3. Luego en `views.py` :

```
from django.conf import settings
from rest_framework.views import APIView

class MyView(APIView):
    queryset = OurModel.objects.all()
    serializer_class = OurModelSerializer
    pagination_class = settings.DEFAULT_PAGINATION_CLASS # cool trick right? :)

    # We need to override get method to achieve pagination
    def get(self, request):
        ...
        page = self.paginate_queryset(self.queryset)
        if page is not None:
            serializer = self.serializer_class(page, many=True)
            return self.get_paginated_response(serializer.data)

        ... Do other stuff needed (out of scope of pagination)

    # Now add the pagination handlers taken from
    #  django-rest-framework/rest_framework/generics.py

    @property
    def paginator(self):
        """
        The paginator instance associated with the view, or `None`.
        """
         if not hasattr(self, '_paginator'):
             if self.pagination_class is None:
                 self._paginator = None
             else:
                 self._paginator = self.pagination_class()
         return self._paginator

     def paginate_queryset(self, queryset):
         """
         Return a single page of results, or `None` if pagination is disabled.
         """
         if self.paginator is None:
             return None
         return self.paginator.paginate_queryset(queryset, self.request, view=self)

     def get_paginated_response(self, data):
         """
         Return a paginated style `Response` object for the given output data.
         """
         assert self.paginator is not None
         return self.paginator.get_paginated_response(data)
```

Ahora tenemos un `APIView` que maneja la paginación!

[Intermedio] Paginación en una vista basada en función

Hemos visto en esos ejemplos (ex_1 , ex_2) cómo usar y anular las clases de paginación en cualquier vista base de clase genérica.
¿Qué sucede cuando queremos usar la paginación en una vista basada en funciones?

Supongamos también que queremos crear una vista basada en funciones para `MyModel` con `PageNumberPagination` , respondiendo solo a una solicitud `GET` . Entonces:

```
from rest_framework.pagination import PageNumberPagination


@api_view(['GET',])
def my_function_based_list_view(request):
    paginator = PageNumberPagination()
    query_set = MyModel.objects.all()
    context = paginator.paginate_queryset(query_set, request)
    serializer = MyModelSerializer(context, many=True)
    return paginator.get_paginated_response(serializer.data)
```

Podemos hacer lo anterior para una paginación personalizada cambiando esta línea:

```
paginator = PageNumberPagination()
```

a esto

```
paginator = MyCustomPagination()
```

siempre que hayamos definido `MyCustomPagination` para anular alguna paginación predeterminada

Lea Paginación en línea: https://riptutorial.com/es/django-rest-framework/topic/9950/paginacion

Capítulo 9: Serializadores

Examples

Acelerar las consultas de serializadores

Digamos que tenemos un modelo de `Travel` con muchos campos relacionados:

```
class Travel(models.Model):

    tags = models.ManyToManyField(
        Tag,
        related_name='travels', )
    route_places = models.ManyToManyField(
        RoutePlace,
        related_name='travels', )
    coordinate = models.ForeignKey(
        Coordinate,
        related_name='travels', )
    date_start = models.DateField()
```

Y queremos construir CRUD en `/travels` través de ViewSet.
Aquí está el simple conjunto de vistas:

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()
    serializer_class = TravelSerializer
```

El problema con este ViewSet es que tenemos muchos campos relacionados en nuestro modelo de `Travel` , por lo que Django golpeará db para cada instancia de `Travel` . Podemos llamar a select_related y prefetch_related directamente en el atributo `queryset` , pero si queremos separar los serializadores para la `list` , `retrieve` , `create` ... las acciones de ViewSet.
Entonces podemos poner esta lógica en una mezcla y heredarla:

```
class QuerySerializerMixin(object):
    PREFETCH_FIELDS = [] # Here is for M2M fields
    RELATED_FIELDS = [] # Here is for ForeignKeys

    @classmethod
    def get_related_queries(cls, queryset):
        # This method we will use in our ViewSet
        # for modify queryset, based on RELATED_FIELDS and PREFETCH_FIELDS
        if cls.RELATED_FIELDS:
            queryset = queryset.select_related(*cls.RELATED_FIELDS)
        if cls.PREFETCH_FIELDS:
            queryset = queryset.prefetch_related(*cls.PREFETCH_FIELDS)
        return queryset


    class TravelListSerializer(QuerySerializerMixin, serializers.ModelSerializer):
```

```
        PREFETCH_FIELDS = ['tags'']
        RELATED_FIELDS = ['coordinate']
        # I omit fields and Meta declare for this example


    class TravelRetrieveSerializer(QuerySerializerMixin, serializers.ModelSerializer):

        PREFETCH_FIELDS = ['tags', 'route_places']
```

Ahora reescriba nuestro `ViewSet` con nuevos serializadores

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()

    def get_serializer_class():
        if self.action == 'retrieve':
            return TravelRetrieveSerializer
        elif self.action == 'list':
            return TravelListSerializer
        else:
            return SomeDefaultSerializer


    def get_queryset(self):
        # This method return serializer class
        # which we pass in class method of serializer class
        # which is also return by get_serializer()
        q = super(TravelViewset, self).get_queryset()
        serializer = self.get_serializer()
        return serializer.get_related_queries(q)
```

Serializadores anidados actualizables

Los serializadores anidados de forma predeterminada no admiten la creación y actualización. Para admitir esto sin duplicar la lógica de creación / actualización de DRF, es importante *eliminar* los datos anidados de `validated_data` antes de delegar a `super` :

```
# an ordinary serializer
class UserProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserProfile
        fields = ('phone', 'company')


class UserSerializer(serializers.ModelSerializer):
    # nest the profile inside the user serializer
    profile = UserProfileSerializer()

    class Meta:
        model = UserModel
        fields = ('pk', 'username', 'email', 'first_name', 'last_name')
        read_only_fields = ('email', )

    def update(self, instance, validated_data):
        nested_serializer = self.fields['profile']
        nested_instance = instance.profile
```

```
        # note the data is `pop`ed
        nested_data = validated_data.pop('profile')
        nested_serializer.update(nested_instance, nested_data)
        # this will not throw an exception,
        # as `profile` is not part of `validated_data`
        return super(UserDetailsSerializer, self).update(instance, validated_data)
```

En el caso de `many=True` , Django se quejará de que `ListSerializer` no admite la `update` . En ese caso, debe manejar la semántica de la lista usted mismo, pero todavía puede delegar a `nested_serializer.child` .

Orden de Validación del Serializador

En DRF, la validación del serializador se ejecuta en un orden específico no documentado

1. `field.run_validators` campo llamada (`serializer.to_internal_value` and `field.run_validators`)
2. `serializer.validate_[field]` se llama para cada campo.
3. Se llaman los validadores de nivel de `serializer.run_validation` (`serializer.run_validation` seguido de `serializer.run_validators`)
4. Finalmente, se llama a `serializer.validate` para completar la validación.

Obtención de una lista de todos los objetos secundarios relacionados en el serializador de padres

Supongamos que implementamos una API simple y tenemos los siguientes modelos.

```
class Parent(models.Model):
    name = models.CharField(max_length=50)


class Child(models.Model):
    parent = models.ForeignKey(Parent)
    child_name = models.CharField(max_length=80)
```

Y queremos devolver una respuesta cuando un `parent` particular se recupera a través de la API.

```
{
'url': 'https://dummyapidomain.com/parents/1/',
'id': '1',
'name': 'Dummy Parent Name',
'children': [{
        'id': 1,
        'child_name': 'Dummy Children I'
    },
    {
        'id': 2,
        'child_name': 'Dummy Children II'
    },
    {
        'id': 3,
        'child_name': 'Dummy Children III'
    },
```

```
    ...
],
```

}

Para ello, implementamos los serializadores correspondientes de esta forma:

```
class ChildSerializer(serializers.HyperlinkedModelSerializer):

    parent_id =
serializers.PrimaryKeyRelatedField(queryset=Parent.objects.all(),source='parent.id')

    class Meta:
        model = Child
        fields = ('url','id','child_name','parent_id')

    def create(self, validated_data):
        subject = Child.objects.create(parent=validated_data['parent']['id'],
child_name=validated_data['child_name'])

        return child


class ParentSerializer(serializers.HyperlinkedModelSerializer):
    children = ChildSerializer(many=True, read_only=True)
    class Meta:
        model = Course
        fields = ('url','id','name','children')
```

Para que esto funcione correctamente aplicación necesitamos actualizar nuestro `Child` modelo y añadir un **related_name** a `parent` campo. La versión actualizada de nuestra implementación de modelo `Child` debería ser así:

```
class Child(models.Model):
parent = models.ForeignKey(Parent, related_name='children')  # <--- Add related_name here
child_name = models.CharField(max_length=80)
```

Al hacer esto, podremos obtener la lista de todos los objetos secundarios relacionados en el serializador principal.

Lea Serializadores en línea: https://riptutorial.com/es/django-rest-framework/topic/2377/serializadores

Capítulo 10: Serializadores

Introducción

De acuerdo con la documentación oficial de DRF, los serializadores ayudan a convertir datos complejos como querysets y modelos de instancia a tipos de datos nativos de python para que puedan procesarse como JSON, XML y otros tipos de contenido.

Los serializadores en DRF son más parecidos a las clases django **Form** y **ModelForm** . **La** clase **Serializer** nos proporciona una forma personalizada de manejar los datos como django Form. Y la clase **ModelSerializer** nos proporciona una manera fácil de manejar datos basados en modelos.

Examples

Introducción Básica

Digamos que tenemos un modelo llamado producto.

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.IntegerField()
```

Ahora vamos a declarar un modelo de serializadores para este modelo.

```
from rest_framework.serializers import ModelSerializer

class ProductSerializers(ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
        read_only_fields = ('id',)
```

Por esta clase de ProductSerializers hemos declarado un modelo de serializadores. En la clase Meta, por variable de modelo, le dijimos al ModelSerializer que nuestro modelo será el modelo de Producto. Por campos variables dijimos que este serializador debería incluir todo el campo del modelo. Por último, mediante la variable read_only_fields, dijimos que id será un campo de 'solo lectura', que no se puede editar.

Veamos que hay en nuestro serializador. Al principio, importe el serializador en la línea de comandos y cree una instancia y luego imprímala.

```
>>> serializer = ProductSerializers()
>>> print(serializer)
ProductSerializers():
        id = IntegerField(label='ID', read_only=True)
        name = CharField(max_length=100)
        price = IntegerField(max_value=2147483647, min_value=-2147483648)
```

Por lo tanto, nuestro serializador toma todo el campo de nuestro modelo y crea todo el campo a su manera.

Podemos usar ProductSerializers para serializar un producto, o una lista de productos.

```
>>> p1 = Product.objects.create(name='alu', price=10)
>>> p2 = Product.objects.create(name='mula', price=5)
>>> serializer = ProductSerializers(p1)
>>> print(serializer.data)
{'id': 1,'name': 'alu', 'price': 10}
```

En este punto, hemos traducido la instancia del modelo a tipos de datos nativos de Python. Para finalizar el proceso de serialización, procesamos los datos en json.

```
>>> from rest_framework.renderers import JSONRenderer
>>> serializer = ProductSerializers(p1)
>>> json = JSONRenderer().render(serializer.data)
>>> print(json)
'{"id": 1,"name": "alu", "price": 10}'
```

Lea Serializadores en línea: https://riptutorial.com/es/django-rest-framework/topic/8871/serializadores

Capítulo 11: Uso de django-rest-framework con AngularJS como framework de front-end.

Introducción

En este tema veremos cómo usar Django REST Framework como una API de back-end para una aplicación AngularJS. Los principales problemas que surgen entre el uso de DRF y AngularJS en general giran en torno a la comunicación HTTP entre las dos tecnologías, así como a la representación de los datos en ambos extremos y, finalmente, a cómo implementar y diseñar la aplicación / sistema.

Examples

Vista DRF

```
class UserRegistration(APIView):
    def post(self, request, *args, **kwargs):
        serializer = UserRegistrationSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.to_representation(instance=serializer.instance),
status=status.HTTP_201_CREATED)
```

Solicitud angular

```
$http.post("\<domain\>/user-registration/", {username: username, password: password})
     .then(function (data) {
         // Do success actions here
     });
```

Lea Uso de django-rest-framework con AngularJS como framework de front-end. en línea: https://riptutorial.com/es/django-rest-framework/topic/10893/uso-de-django-rest-framework-con-angularjs-como-framework-de-front-end-

Creditos

S. No	Capítulos	Contributors
1	Empezando con django-rest-framework	Abhishek, Chitrank Dixit, Community, davyria, itmard, Majid, Rahul Gupta, Saksow, ssice
2	Autenticación	iankit, itmard
3	Autenticación de tokens con Django Rest Framework	Ali_Waris, Cadmus
4	Enrutadores	John Moutafis
5	Filtros	Bitonator
6	Mixins	John Moutafis
7	Paginación	John Moutafis
8	Serializadores	hnroot, Ivan Semochkin, Louis Barranqueiro, Silly Freak
9	Uso de django-rest-framework con AngularJS como framework de front-end.	mattjegan