

Servicios REST

con Django REST Framework

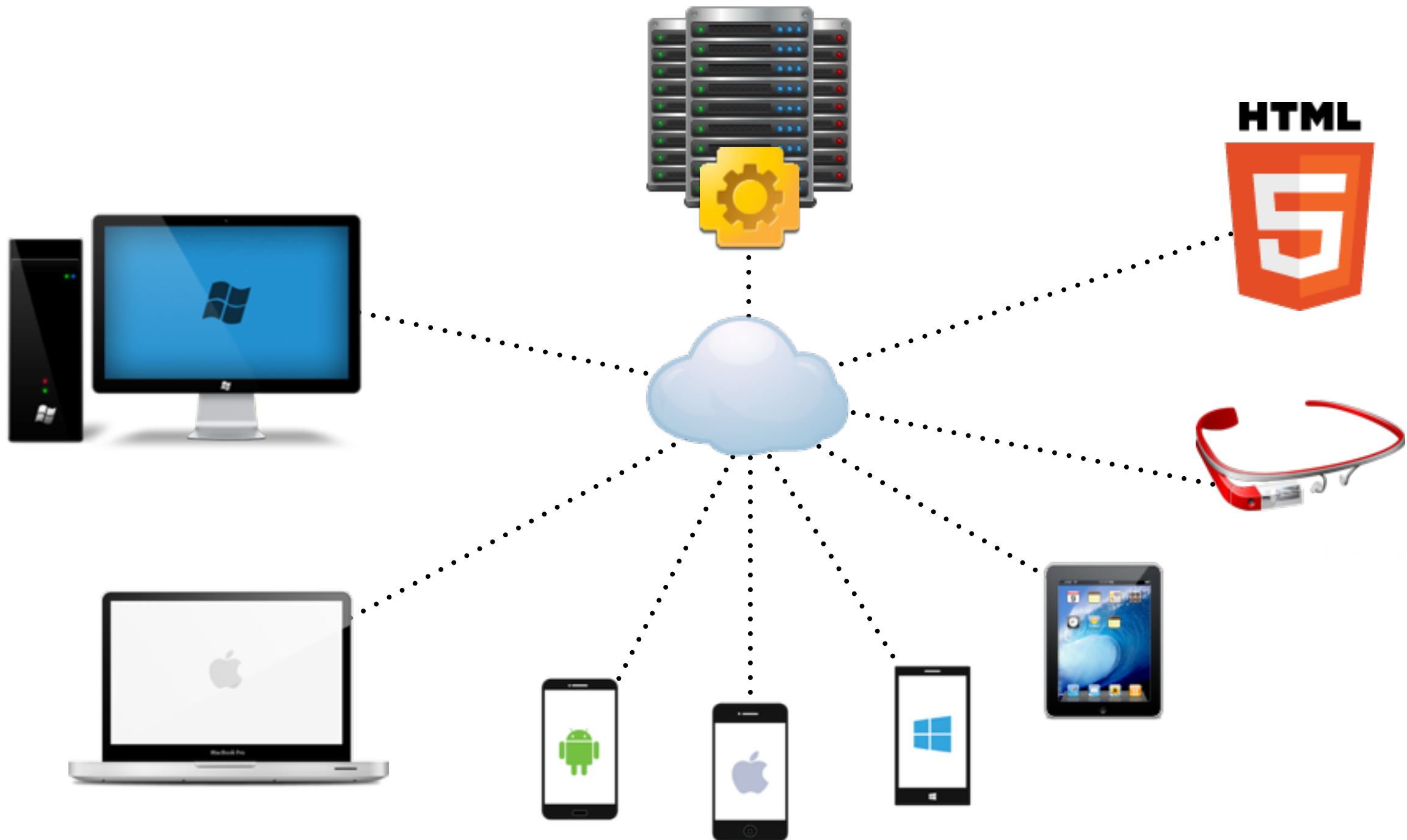
¿Qué es REST?

- **RE**presentational **S**tate **T**ransfer
- Es una técnica de arquitectura de software
- La idea: utilizar HTTP para comunicar dispositivos
- Las aplicaciones que usan REST, se llaman RESTful
- Con REST tenemos WebServices más ligeros y simples
- A los WebServices, los llamaremos APIs RESTful

¿Quién usa REST?

- Twitter
- Flickr
- Amazon S3
- Google Glass API (Mirror API)

¿Por qué usar REST?



¿Cómo funciona?

- Usando los propios métodos de HTTP para operaciones CRUD
 - Create — POST
 - Recover — GET
 - Update — PUT
 - Delete — DELETE
- Devuelven como respuesta códigos HTTP y, a veces, contenido
- Un API Rest está compuesta por varios **endpoints**
- **Cada URL del API, tiene un endpoint por método soportado**

GET

Petición

GET /1.1/friends/

Respuesta

- Código 200 - OK
- [{id : 311416, name : "Luke Skywalker"}, {id : 311417, name : "Obi Wan Kenobi"}, {id : 311418, name : "Chewbacca"}, {id : 311419, name : "R2D2"}, {id : 311419, name : "Yoda"}]

GET

Petición

GET /1.1/friends/?count=2&race=jedi

Respuesta

- Código 200 - OK
- [{id : 311416, name : "Luke Skywalker"}, {id : 311417, name : "Obi Wan Kenobi"}]

GET

Petición

GET /1.1/friends/31416

Respuesta

- Código 200 - OK
- {id : 311416, name : “Luke Skywalker”, weapon : “Light saber”, friends : [{id : 311418, name : “Yoda”}, ...] }

POST

Petición

POST /1.1/friends/add

{id : 311418, name : "Yoda"}

Respuesta

- Código 201 - Created
- {id : 311418, name : "Yoda", weapon : "Light saber", friends : [{id : 311416, name : "Luke Skywalker"}, ...] }

PUT

Petición

PUT /1.1/friends/31418

{name : "Master Yoda"}

Respuesta

- Código 202 - Accepted
- {id : 311418, name : "Master Yoda", weapon : "Light saber", friends : [{id : 311416, name : "Luke Skywalker"}, ...] }

DELETE

Petición

DELETE /1.1/friends/31416

Respuesta

- Código 204 - No Content

Códigos HTTP

- 1xx - Códigos de información
- 2xx - Códigos de éxito
- 3xx - Códigos de redirección
- 4xx - Códigos de error de cliente
- 5xx - Códigos de error de servidor

Códigos 2xx

- 200 - OK
- 201 - Created
- 202 - Accepted
- 203 - Non-Authoritative Information
- 204 - No Content

Códigos 2xx

- 205 - Reset Content
- 206 - Partial Content
- 207 - Multi-Status
- 208 - Already Reported
- 226 - IM Used

Códigos 3xx

- 300 - Multiple Choices
- 301 - Moved Permanently
- 302 - Found
- 303 - See Other
- 304 - Not modified

Códigos 3xx

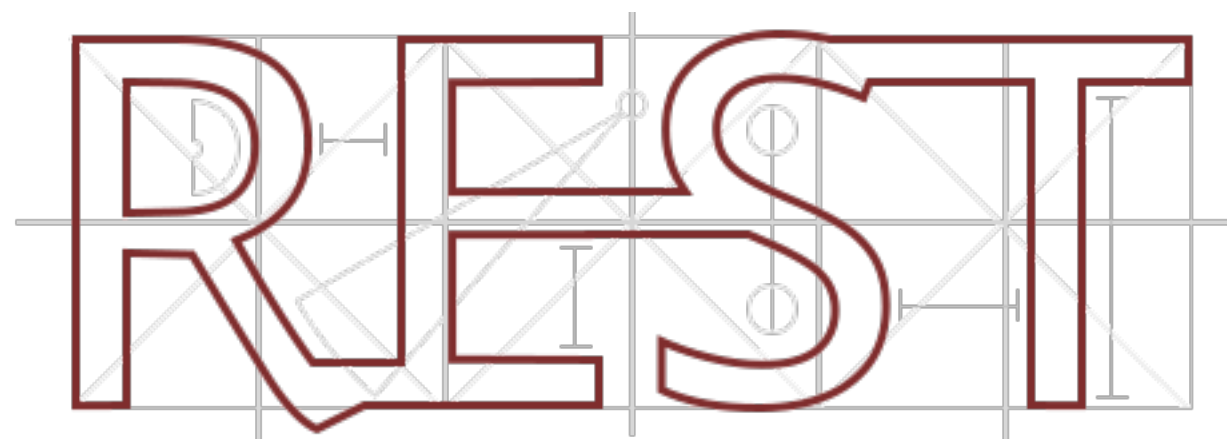
- 305 - Use proxy
- 306 - Switch proxy
- 307 - Temporary Redirect
- 308 - Permanent Redirect

Códigos 4xx

- 400 - Bad Request
- 401 - Unauthorized
- 402 - Payment Required
- 403 - Forbidden
- 404 - Not found
- 405 - Method Not Allowed

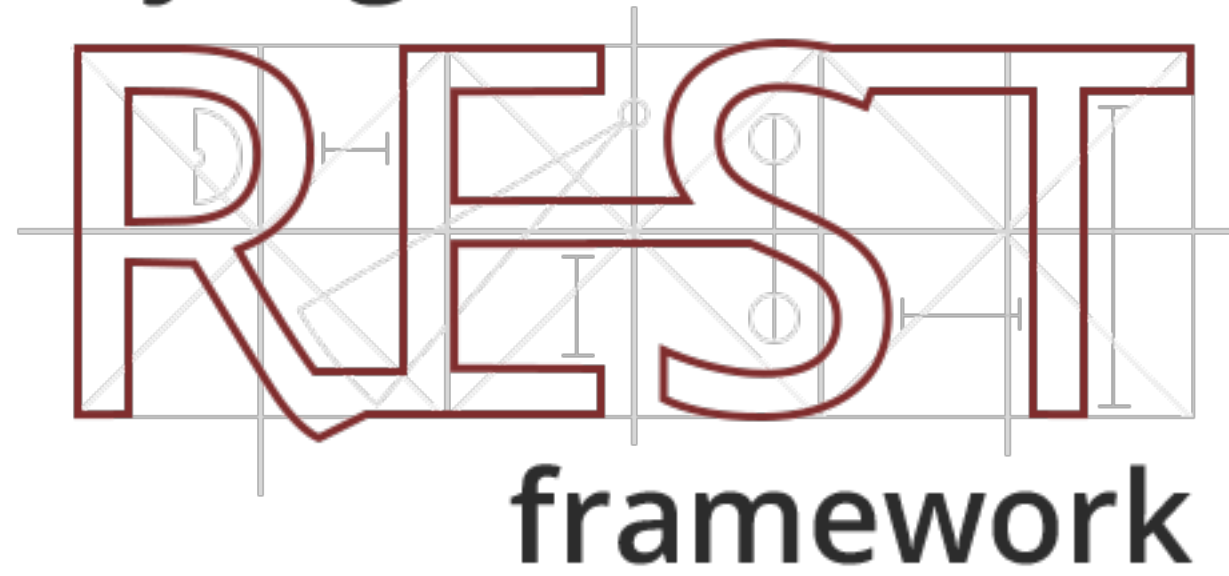
Códigos 4xx

- 406 - Not Acceptable
- 407 - Proxy Authentication Failed
- 408 - Request Timeout
- 409 - Conflict
- 410 - Gone
- ...



django

REST



framework

Cosas que lo hacen
tan maravilloso

Cosas que lo hacen tan maravilloso

- Utiliza el mismo sistema de vistas que Django
- Proporciona un API navegable, muy útil para los que integran con el API
- Utilidades de autenticación como OAuth1 y OAuth2
- Serialización de recursos basados en ORM y no-ORM

Cosas que lo hacen tan maravilloso

- Totalmente personalizable
- Muy bien documentado
- Usado por empresas como Mozilla y Eventbrite

Instalando Django REST Framework

Instalando Django REST Framework

```
Frikr$ source env/bin/activate
```

```
(env)Frikr$ pip install djangorestframework
```

Instalando Django REST Framework

En Windows

```
C:/.../Frikr> env/Scripts/activate
```

```
(env)C://.../Frikr> pip install djangorestframework
```

Instalando Django REST Framework

Añadirlo a las apps instaladas en settings.py:

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

Serializers

Nuestros intérpretes

Serializers

- Los serializers se encargan de transformar los datos de una petición HTTP en un objeto y viceversa.
- REST Framework se encarga de transformar los datos en función de su formato (JSON, XML, etc.) en un diccionario que pasa al serializer.
- Se definen como los modelos de Django (con Fields)

Serializers

Hay que definir un método `restore_object` para construir el objeto en base al diccionario con los datos:

```
restore_object(self, attrs, instance=None)
```

- `attrs` es el diccionario con los datos serializados
- `instance` es el objeto que queremos actualizar con los datos que viene en el diccionario (y puede no existir)

Vistas de Django REST Framework

Vistas de Django REST Framework

- Al igual que Django proporciona las View, Django REST Framework proporciona las APIView
- Estas vistas heredan de la clase View de Django
- Modifican el objeto request, eliminando los atributos GET y POST y centralizándolo todo en un atributo DATA
- Deserializan los datos en el formato que se envían (JSON, XML, etc.) y los representa como un diccionario en el atributo DATA

Vistas de Django REST Framework

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

```
class PlanetList(APIView):

    def get(self, request, format=None):
        [...]
        return Response(serializer.data)

    def post(self, request, format=None):
        [...]
        return Response(serializer.data)

    def put(self, request, format=None):
        [...]
        return Response(serializer.data)

    def delete(self, request, format=None):
        [...]
        return Response(serializer.data)
```

ModelSerializers

ModelSerializers

- Los ModelSerializers nos permiten crear Serializers a partir de modelos Django
- Nos ahorran implementar el método `restore_object`

ModelSerializers

```
from rest_framework import serializers
```

```
from models import WallMessage
```

```
class WallMessageSerializer(serializers.ModelSerializer):
```

```
    user = serializers.Field(source='user.username')
```

```
    class Meta:
```

```
        model = WallMessage
```

```
        fields = ('id', 'user', 'message', 'created_at')
```

Vistas genéricas

Vistas genéricas

- Django REST Framework proporciona vistas genéricas para operaciones habituales:
 - CreateAPIView: endpoint de creación (POST)
 - ListAPIView: endpoint de listado de elementos (GET)
 - RetrieveAPIView: endpoint de detalle de un elemento (GET)
 - DestroyAPIView: endpoint de borrado de un elemento (DELETE)
 - UpdateAPIView: endpoint de actualización de elemento (UPDATE)
 - ListCreateAPIView: endpoint de listado y creación de elementos (GET, POST)
 - RetrieveUpdateAPIView: endpoint de detalle y actualización de un elemento (GET, UPDATE)
 - RetrieveDestroyAPIView: endpoint de detalle y borrado de un elemento (GET, DELETE)
 - RetrieveUpdateDestroyAPIView: endpoint de detalle, actualización y borrado de un elemento (GET, UPDATE, DELETE)

Vistas genéricas

```
from rest_framework import generics
```

```
class PlanetList(generics.ListCreateAPIView):
```

```
    queryset = Planet.objects.all()
```

```
    serializer_class = PlanetSerializer
```

```
class PlanetDetail(generics.RetrieveUpdateDestroyAPIView):
```

```
    queryset = Planet.objects.all()
```

```
    serializer_class = PlanetSerializer
```

Validación de objetos

La validación, siempre en los Serializers

Validación de objetos

- A nivel de atributo definiendo un método:
 - **validate_<ATRIBUTO>(self, attrs, source)**
 - attrs: diccionario con atributos
 - source: el nombre del atributo
 - Debe devolver los attrs o lanzar una excepción `serializers.ValidationError`
- A nivel de objeto definiendo el método:
 - **validate(self, attrs)**
 - attrs: diccionario con atributos
 - Debe devolver los attrs o lanzar una excepción `serializers.ValidationError`

Autenticación y autorización

Autenticación y autorización

- Django REST Framework nos proporciona una serie de utilidades para trabajar con la autenticación y autorización (permisos)
- Autenticación: nos permite saber quién está usando el API
- Autorización: nos permite definir si el usuario que está usando el API, está autorizado a hacer la acción que solicita o no

Autenticación

- Con Django, sabemos qué usuario está autenticado gracias al atributo **user** del objeto request que recibe cada vista.

Autorización (permisos)

- En nuestros controladores podemos añadir una clase para gestionar la autorización (o permisos) de cada operación.
- Podemos utilizar clases predefinidas o bien crear nuestra propia clase para gestionar la autorización.

Autorizaciones genéricas

- **AllowAny:** permite todo a todos
- **IsAuthenticated:** permite todo a los usuarios autenticados
- **IsAdminUser:** sólo se permite hacer cosas a usuarios admin
- **IsAuthenticatedOrReadOnly:** permite operaciones de lectura a todos y de escritura/actualización/borrado a usuarios autenticados
- **DjangoModelPermissions:** utiliza los modelos de permisos de Django
- **DjangoModelPermissionsOrAnonReadOnly:** utiliza los modelos de permisos de Django para operaciones de escritura/actualización/borrado con usuarios autenticados y permite operaciones de lectura a cualquiera
- **DjangoObjectPermissions:** utiliza el framework de django de objects permissions.
- **TokenHasReadWriteScope:** para utilizar con autenticación OAuth u OAuth2

Autorizaciones personalizadas

- Debemos crear una clase que hereda de
 - `rest_framework.permissions.BasePermission`
- Implementar los métodos `has_permission` y `has_object_permission` que deben devolver `True` o `False`
- El orden de ejecución es:
 1. `has_permission(self, request, view)`
 2. `has_object_permission(self, request, view, obj)`

Autorizaciones personalizadas

```
from rest_framework import permissions
```

```
class PlanetPermissions(permissions.BasePermission):
```

```
    def has_object_permission(self, request, view, obj):
```

```
        return request.user.is_superuser
```

```
    def has_permission(self, request, view):
```

```
        # se ejecuta sólo en peticiones PUT y DELETE
```

```
        return request.user.is_authenticated()
```


Paginación de listados

Paginación de listados

- Django REST Framework nos facilita la paginación de resultados en los endpoints de listado
- Tan sólo hay que añadir los datos de paginación al diccionario de configuración de REST Framework en el settings.py del proyecto

Paginación de listados

```
REST_FRAMEWORK = {
```

```
    'PAGINATE_BY': 5, # 5 items por página
```

```
    'PAGINATE_BY_PARAM': 'page_size', # parámetro  
    GET para definir el número de elementos por página
```

```
    'MAX_PAGINATE_BY': 10 # parámetro GET para  
    definir el máximo número de elementos por página
```

```
}
```

ViewSet

Haciendo todavía menos

ViewSet

- Con los ViewSets nos ahorramos tener que escribir diferentes las vistas de listado y de detalle.
- Proporcionan los métodos:
 - list # GET
 - create # POST
 - retrieve # RETRIEVE
 - update # UPDATE
 - partial_update # PATCH
 - destroy # DELETE

ViewSets

- Tienen un atributo muy útil llamado “action” que permite conocer en todo momento que acción se está realizando (list, create, retrieve, update, delete).
- Hay también unos ViewSets genéricos:
 - ModelViewSet: indicando un ModelSerializer y un queryset, se encarga de todo
 - ReadOnlyModelViewSet: indicando un ModelSerializer y un queryset, proporciona un API de lectura completa

ViewSet

```
class PlanetViewSet(viewsets.ModelViewSet):  
    queryset = Planet.objects.all()  
    serializer_class = PlanetDetailSerializer  
    permission_classes = (PlanetPermissions,)
```

Routers

Routers

- Los Routers se encargan de generar automáticamente las URL de nuestra API
- Ejemplo:
 - /planets/ # para el listado de planetas
 - /planets/1 # para el detalle

Routers

```
router = DefaultRouter()
```

```
router.register(r'planets', api.PlanetViewSet)
```

```
urlpatterns = patterns("",
```

```
    url(r'^', include(router.urls)),
```

```
)
```

Filtrado, ordenación y búsqueda

Filtrado, ordenación y búsqueda

- Es bastante habitual ofrecer formas de filtrar y ordenar datos en los endpoint de listados
- Se admiten una serie de parámetros GET para realizar filtrados
- Podemos automatizar el filtrado y ordenación de listados de modelos en función de los campos de los mismos utilizando **rest_framework.filters**

Filtrado, ordenación y búsqueda

- Tenemos que añadir el atributo **filter_backends** a la vista y asignar los backends de filtrado, ordenación y búsqueda:
- Búsqueda: añadir *rest_framework.filters.SearchFilter* a *filter_backends* e indicar los campos en los que se busca en *search_filters*
- Ordenación: añadir *rest_framework.filters.OrderingFilter* a *filter_backends* e indicar los campos en los que se permite ordenar *ordering_filters*

Filtrado, ordenación y búsqueda

- Para realizar búsquedas desde el cliente, hay que utilizar el parámetro GET *search* e indicar la cadena de búsqueda
Ejemplo: `/flights/?search=Lisboa`
- Para ordenación, hay que utilizar el parámetro GET *ordering* e indicar los campos de ordenación (puede ser varios separados por comas). Si queremos ordenación inversa, podemos poner el nombre del campo con el símbolo “-” delante.
Ejemplo: `/users/?ordering=first_name,-last_name`

Filtrado, ordenación y búsqueda

- Para poder realizar filtrados por campos:
 - Hay que instalar la librería django-filter
 - Utilizar *DjangoFilterBackend* como backend de filtrado
 - Indicar el *filter_fields* los campos por los que se permite filtrar

Filtrado, ordenación y búsqueda

<http://www.django-rest-framework.org/api-guide/filtering>

Respondiendo en
otros formatos

Respondiendo en otros formatos

- Por defecto, Django REST Framework responde al usuario en JSON o con el API navegable.
- Es posible configurar otros formatos de respuesta, como XML o YAML
- Simplemente tenemos que añadir los backends de renderizado en el diccionario de configuración de REST Framework en el settings.py del proyecto
- En las peticiones, el cliente deberá utilizar la cabecera HTTP Accept para definir en qué formato quiere recibir los datos.

Respondiendo en otros formatos

```
REST_FRAMEWORK = {  
    'DEFAULT_RENDERER_CLASSES': (  
        'rest_framework.renderers.JSONRenderer', # para JSON  
        'rest_framework.renderers.XMLRenderer', # para XML  
        'rest_framework.renderers.BrowsableAPIRenderer' # para  
API navegable  
    )  
}
```

Subiendo archivos

Subiendo archivos

- La subida de archivos en un API REST con Django REST Framework se implementa como un endpoint normal.
- Lo más sencillo es utilizar un modelo con un campo FileField, su ModelSerializer y una Generic View
- El “truco” está en el envío de los datos, no se puede utilizar un formato RAW, hay que utilizar un formato “form-data” donde el parámetro donde se envía el fichero tenga el mismo nombre que el FileField del modelo

Modelo

```
class File(models.Model):
```

```
    file = models.FileField(upload_to='uploads')
```

ModelSerializer

```
class FileSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        model = File
```

Generic View

```
class PhotoUploadAPI(CreateAPIView):  
    queryset = File.objects.all()  
  
    serializer_class = FileSerializer  
  
    permission_classes = (IsAuthenticated,) # opcional  
  
    # pero no es buena idea permitir subir archivos a  
    cualquiera
```