



Proyecto 2

Servicio web REST

Integrantes

Gonzalo Salinas

Ignacio Valdes

Camila Carrasco

Asignatura

Computación paralela y distribuida

Docente

Sebastian Salazar

Entrega

Jueves, 06 de Agosto de 2020

Introducción

La transferencia de estado representacional (en inglés representational state transfer) o REST es un estilo de arquitectura software para sistemas hipertexto distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo. El presente trabajo tiene como propósito comprender el funcionamiento del protocolo HTTP en utilización del REST para realizar la interoperabilidad entre aplicaciones web. Este conocimiento nos permitirá proponer un servicio web REST que resuelva una necesidad específica de la Universidad Tecnológica Metropolitana, que en este caso es, una aplicación que entregue en función de los puntajes obtenidos por el estudiante, las 10 carreras en las que tiene mayores opciones de ingreso desde la que tiene mejor opción hasta la que tiene menor opción.

Definición Servicio Web

Un Servicio Web es un servicio de negocio programado en algún lenguaje, que envuelve cierta funcionalidad y la expone en una red de manera estandarizada, para que cualquiera conectado a esta misma red, que además soporte dichos estándares, pueda acceder a ella. Generalmente, la interacción se basa en el envío de solicitudes y respuestas entre un cliente y un servidor, que incluyen datos. El cliente solicita información, enviando a veces datos al servidor para que pueda procesar su solicitud. El servidor genera una respuesta que envía de vuelta al cliente, adjuntando otra serie de datos que forman parte de esa respuesta. Por tanto, podemos entender un servicio web como un tráfico de mensajes entre dos máquinas.

Servicio Web REST

REST es una arquitectura para aplicaciones basadas en redes, sus siglas significan REpresentational State Transfer y utiliza el protocolo HTTP para el manejo de información y datos. A diferencia de un servicio SOAP, el REST presenta una característica mucho más flexible a la hora de transportar datos, ya que a diferencia del primero, no solo se maneja con el envío de XML, si no que también puede transmitir (en base al header content-type) json, binarios, text, xml, etc.

La gran mayoría de los servicios REST utilizan el formato de envío de datos json, y esto se debe principalmente a la interpretación de forma nativa por JavaScript de este, así, frameworks como Angular y React se aprovechen al máximo, pues pueden enviar peticiones directas al servidor por medio de AJAX y obtener los datos de una forma nativa.

Tecnología utilizada

Para el desarrollo del REST solicitado se utilizará DJANGO, un framework de desarrollo web de código abierto escrito en python que se caracteriza por la facilidad de su utilización y su versatilidad a la hora de crear un web service, orientando su estructuración a objetos y clases. También se debe destacar la interfaz que presenta por defecto en el navegador, el cual ayuda a la realización de pruebas y permite una visualización más aproximada al resultado final sin la necesidad de programar una sola línea de código para un front básico de testeo (en caso de que se requiera crear uno), es importante destacar esta característica no por su primera necesidad, ya que no la tiene, si no por una interesante oferta de comodidad para el desarrollador que django ofrece. Otro punto importante para la situación de este proyecto que django también permite, es el manejo de bases de datos orientado a objetos, la cual se estructurará utilizando SQLite. Para la realización de pruebas y emulación del cliente que consumirá el servicio, se utilizará el software POSTMAN.

Instalación de Requerimientos

Para la utilización del REST es necesaria la instalación de **python 3 y pip en su última versión**, luego, se deben instalar las librerías necesarias para su funcionamiento, para esto, se debe situar la consola de comandos **en la carpeta rest**, y ejecutar las siguientes líneas de código:

Para Windows:

<code>pip install virtualenv</code>	<code>#Instalación de virtualenv</code>
<code>virtualenv -p python3 "env"</code>	<code>#Para crear el entorno virtual</code>

Luego:

<code>source env/Scripts/activate</code>	<code>#Para activar el entorno virtual</code>
<code>pip install -r requisitos.txt</code>	<code>#Instala todos los paquetes necesarios en el entorno</code>

o en su defecto:

<code>pip install virtualenv</code>	<code>#Instalación de virtualenv</code>
<code>virtualenv -p python3 "env"</code>	<code>#Para crear el entorno virtual</code>
<code>cd env/Scripts</code>	<code>#Para ubicar la carpeta de scripts</code>
<code>activate</code>	<code>#Para activar el entorno virtual</code>
<code>cd..</code>	<code>#Para volver atrás en directorio</code>
<code>cd..</code>	<code>#Para volver atrás en directorio</code>
<code>pip install -r requisitos.txt</code>	<code>#Instala todos los paquetes necesarios en el entorno</code>

Para Linux:

<code>sudo pip3 install virtualenv</code>	<code>#Instalación de virtualenv</code>
<code>virtualenv env</code>	<code>#Para crear el entorno virtual</code>
<code>source env/bin/activate</code>	<code>#Para activar el entorno virtual</code>
<code>pip3 install -r requisitos.txt</code>	<code>#Instala todos los paquetes necesarios en el entorno</code>

Postman:

Además de lo anterior, es necesaria la instalación del programa POSTMAN, el cual se puede descargar siguiendo el enlace: <https://www.postman.com/downloads/> para su instalacion en windows, y en el caso de linux, ejecutar la siguiente línea de código:

<code>sudo snap install postman</code>	<code>#Mediante snap el sistema procederá a instalar postman</code>
--	---

Manual de uso

Para la utilización del servicio REST, se ocupará POSTMAN como emulador del cliente, pero antes, es necesario habilitar el servidor, para esto, sitúese en la carpeta rest y ejecute las siguientes líneas de comando:

Para windows:

<code>source env/Scripts/activate</code>	<code>#Para activar el entorno virtual (si es que no lo esta)</code>
<code>python manage.py runserver</code>	<code>#Ejecuta el servidor</code>

Para Linux:

<code>source env/bin/activate</code>	<code>#Para activar el entorno virtual (si es que no lo esta)</code>
<code>python manage.py runserver</code>	<code>#Ejecuta el servidor</code>

Luego de lo anterior debe aparecer el siguiente log en consola, el cual representa la activación exitosa del servidor:

```
$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 05, 2020 - 18:02:45
Django version 3.0.8, using settings 'proyectorrest.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

En este caso, el servidor se está ejecutando en la dirección <http://127.0.0.1:8000/>, por lo que para acceder a las distintas funcionalidades se deben utilizar los siguientes links:

Para consultar por código de carrera: <http://127.0.0.1:8000/api/v0.01/codigo>

Para consultar por nombre de carrera: <http://127.0.0.1:8000/api/v0.01/carrera>

Para consultar opciones de postulación: <http://127.0.0.1:8000/api/v0.01/postular>

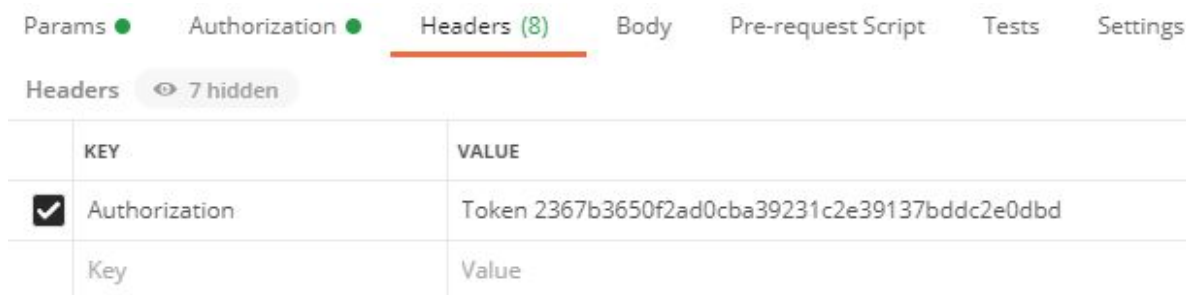
Para consultar los autores del trabajo: <http://127.0.0.1:8000/api/v0.01/autores>

Luego de estos pasos, el trabajo con el servidor está completo, lo que sigue ahora, es la configuración de POSTMAN para consumir el servicio.

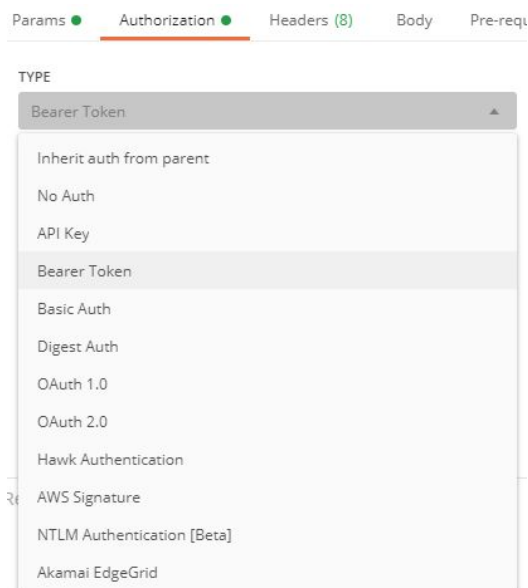
Luego de abrir POSTMAN, lo primero que debe hacer es la configuración de autorización, la cual está basada en un Token por usuario, en este caso solo existe uno el cual se presenta a continuación:

Token 2367b3650f2ad0cba39231c2e39137bddc2e0dbd

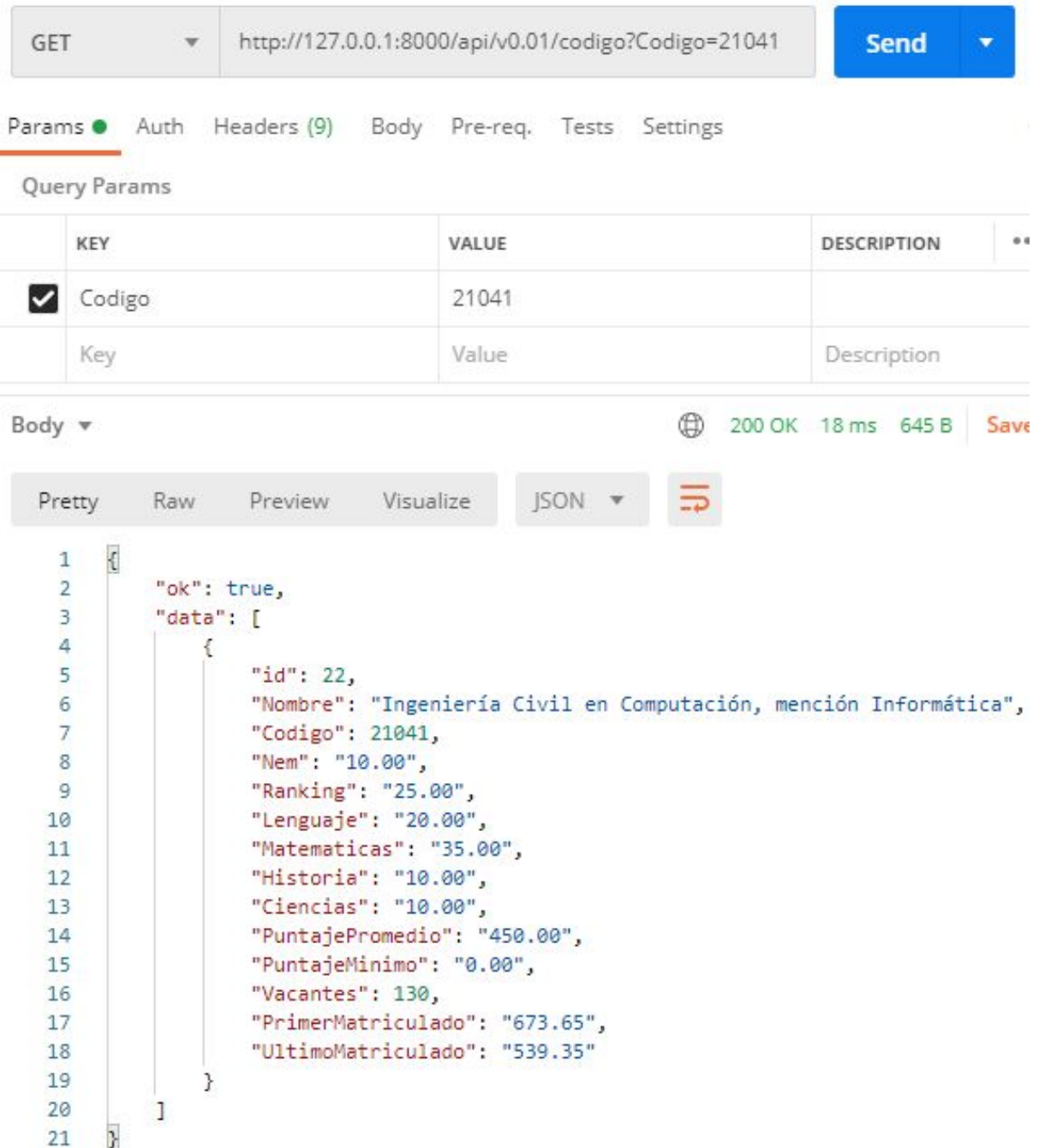
El Token anterior se debe pegar en la **pestaña de headers**, de la siguiente forma:



El siguiente paso es seleccionar el tipo de autorización en la pestaña Authorization, el cual es el Bearer Token:



En esta parte, la casilla de Token se deja en blanco. Luego, un ejemplo de utilización de la operación **CONSULTA POR CÓDIGO** se presenta en la siguiente imagen:



GET http://127.0.0.1:8000/api/v0.01/codigo?Codigo=21041 Send

Params ● Auth Headers (9) Body Pre-req. Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION	**
<input checked="" type="checkbox"/>	Codigo	21041		
	Key	Value	Description	

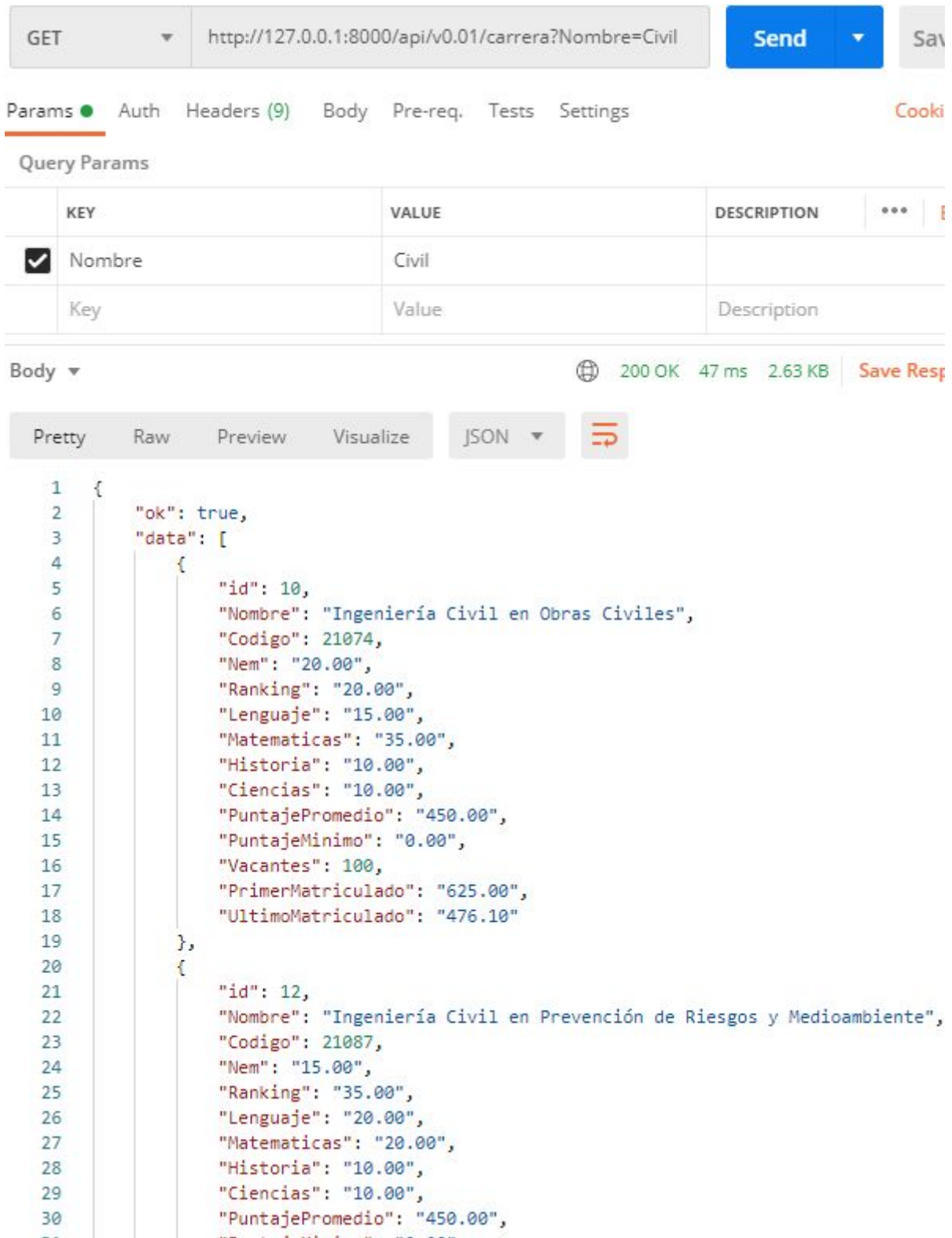
Body ▾ 200 OK 18 ms 645 B Save

Pretty Raw Preview Visualize JSON ↕

```

1  {
2    "ok": true,
3    "data": [
4      {
5        "id": 22,
6        "Nombre": "Ingeniería Civil en Computación, mención Informática",
7        "Codigo": 21041,
8        "Nem": "10.00",
9        "Ranking": "25.00",
10       "Lenguaje": "20.00",
11       "Matematicas": "35.00",
12       "Historia": "10.00",
13       "Ciencias": "10.00",
14       "PuntajePromedio": "450.00",
15       "PuntajeMinimo": "0.00",
16       "Vacantes": 130,
17       "PrimerMatriculado": "673.65",
18       "UltimoMatriculado": "539.35"
19     }
20   ]
21 }
```

Ejemplo de utilización de la operación **CONSULTA POR NOMBRE** se presenta en la siguiente imagen:



GET http://127.0.0.1:8000/api/v0.01/carrera?Nombre=Civil

Send Save

Params Auth Headers (9) Body Pre-req. Tests Settings Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	
<input checked="" type="checkbox"/>	Nombre	Civil			
	Key	Value	Description		

Body 200 OK 47 ms 2.63 KB Save Response

Pretty Raw Preview Visualize JSON

```

1  {
2    "ok": true,
3    "data": [
4      {
5        "id": 10,
6        "Nombre": "Ingeniería Civil en Obras Civiles",
7        "Codigo": 21074,
8        "Nem": "20.00",
9        "Ranking": "20.00",
10       "Lenguaje": "15.00",
11       "Matematicas": "35.00",
12       "Historia": "10.00",
13       "Ciencias": "10.00",
14       "PuntajePromedio": "450.00",
15       "PuntajeMinimo": "0.00",
16       "Vacantes": 100,
17       "PrimerMatriculado": "625.00",
18       "UltimoMatriculado": "476.10"
19     },
20     {
21       "id": 12,
22       "Nombre": "Ingeniería Civil en Prevención de Riesgos y Medioambiente",
23       "Codigo": 21087,
24       "Nem": "15.00",
25       "Ranking": "35.00",
26       "Lenguaje": "20.00",
27       "Matematicas": "20.00",
28       "Historia": "10.00",
29       "Ciencias": "10.00",
30       "PuntajePromedio": "450.00",
31       "PuntajeMinimo": "0.00",
32       "Vacantes": 100,
33       "PrimerMatriculado": "625.00",
34       "UltimoMatriculado": "476.10"
35     }
36   ]
37 }
  
```

Ejemplo de utilización de la operación **MEJORES 10 OPCIONES DE POSTULACIÓN** se presenta en la siguiente imagen:

POST ▼ http://127.0.0.1:8000/api/v0.01/postular Send ▼

Params Auth Headers (11) **Body** ● Pre-req. Tests Settings

form-data ▼

	KEY	VALUE	DESCRIPTION	**
<input checked="" type="checkbox"/>	nem	456		
<input checked="" type="checkbox"/>	lenguaje	560		
<input checked="" type="checkbox"/>	matematicas	605		
<input checked="" type="checkbox"/>	ranking	600		
<input checked="" type="checkbox"/>	ciencias	450		
<input checked="" type="checkbox"/>	historia	750		

Body ▼ 200 OK 32 ms 1.6 KB Sa

Pretty Raw Preview Visualize JSON ▼ ≡

```

1  {
2      "ok": true,
3      "data": [
4          {
5              "codigo_carrera": 21048,
6              "nombre_carrera": "Ingeniería Comercial",
7              "puntaje_postulacion": 627.6,
8              "lugar_tentativo": 22
9          },
10         {
11             "codigo_carrera": 21080,
12             "nombre_carrera": "Ingeniería en Química",
13             "puntaje_postulacion": 611.1,
14             "lugar_tentativo": 1
15         },
16         {
17             "codigo_carrera": 21083,
18             "nombre_carrera": "Química Industrial",
19             "puntaje_postulacion": 611.1,
20             "lugar_tentativo": 1
21         },
22         {
23             "codigo_carrera": 21047.

```


Lógica interna y explicación del funcionamiento

El archivo **settings.py** que está dentro de la **carpeta rest/proyectorrest**, contiene toda la configuración del servicio y el archivo **urls.py** contiene los path de este. Dentro de la **carpeta rest/puntajes** en el archivo **models.py** se puede encontrar el modelo de la base de datos utilizada, el cual es el siguiente:

```
class postulante(models.Model):
    Nombre = models.CharField(max_length = 50)
    Codigo = models.IntegerField()
    Nem = models.DecimalField(max_digits = 5, decimal_places = 2)
    Ranking = models.DecimalField(max_digits = 5, decimal_places = 2)
    Lenguaje = models.DecimalField(max_digits = 5, decimal_places = 2)
    Matematicas = models.DecimalField(max_digits = 5, decimal_places = 2)
    Historia = models.DecimalField(max_digits = 5, decimal_places = 2)
    Ciencias = models.DecimalField(max_digits = 5, decimal_places = 2)
    PuntajePromedio = models.DecimalField(max_digits = 5, decimal_places = 2)
    PuntajeMinimo = models.DecimalField(max_digits = 5, decimal_places = 2)
    Vacantes = models.IntegerField()
    PrimerMatriculado = models.DecimalField(max_digits = 5, decimal_places = 2)
    UltimoMatriculado = models.DecimalField(max_digits = 5, decimal_places = 2)

class autores(models.Model):
    integrante = models.CharField(max_length = 50)
```

Debido a la naturaleza simple de los datos, se crearon solo 2 clases que contienen toda la información necesaria para trabajar, de las cuales, una es la que contiene los datos de los integrantes del equipo de trabajo. Para la utilización de esta información es necesario implementar los serializer de cada clase, dependiendo de las consultas que se requieran para estos datos, que son 3, pero debido a que 2 de ellas utilizan el modelo de la misma forma, se presentan dos serializer para los datos y uno para la clase autores, estos se encuentran en la **carpeta rest/puntajes** en el archivo **serializer.py**:

```
class postulanteSerializer(serializers.ModelSerializer):
    class Meta:
        model = postulante
        fields = '__all__'

class codigoSerializer(serializers.ModelSerializer):
    class Meta:
        model = postulante
        fields = ('Codigo',)

class autoresSerializer(serializers.ModelSerializer):
    class Meta:
        model = autores
        fields = '__all__'
```

Dentro de la **carpeta rest/puntajes** en el archivo **views.py** se encuentran los metodos u operaciones que realiza el servicio con los datos, las cuales se explican a continuación:

Búsqueda por código:

```
27 class buscarcodigo.views.APIView:
28     authentication_classes = [TokenAuthentication]
29     permission_classes = [IsAuthenticated]
30     def get(self, request):
31         if ('Codigo' in request.GET):
32             if (request.GET.get('Codigo').isdigit()==False):
33                 return JsonResponse({'ok':False,'data':"json mal ingresado, la llave 'Codigo' presenta un tipo
34             else:
35                 return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'Codigo'."}, status=400)
36         cod = request.GET.get('Codigo')
37         carrera = postulante.objects.filter(Codigo = cod)
38         serializador = postulanteSerializer(carrera, many=True)
39         return JsonResponse({'ok':True,'data':serializador.data}, status=200)
```

Las líneas 28 y 29 están destinadas al proceso de autenticación y requieren del Token especificado anteriormente para permitir la utilización del método, estas líneas están presentes en todas las clases por lo que no se hablará de ellas nuevamente. Las líneas 30 a 39 componen el método GET, el cual en primera instancia comprueba el correcto ingreso de datos en el json del request (líneas 31 a 35), detectando en primera instancia si las llaves no corresponden y en segunda si los datos están incorrectos, para las excepciones devuelve el código de error HTTP correspondiente y el mensaje especificando el problema. Finalmente, tras la comprobación de todo lo anterior, las líneas 36 a 39 realizan la consulta a la base de datos y entregan la respuesta satisfactoria del método.

Búsqueda por nombre:

```
41 class buscarcarrera.views.APIView:
42     authentication_classes = [TokenAuthentication]
43     permission_classes = [IsAuthenticated]
44     def get(self, request):
45         if ('Nombre' in request.GET):
46             if (request.GET.get('Nombre').isalpha()==False):
47                 return JsonResponse({'ok':False,'data':"json mal ingresado, la llave 'Nombre' presenta un tipo
48             else:
49                 return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'Nombre'."}, status=400)
50         consulta = request.GET.get('Nombre')
51         resultado = postulante.objects.filter(Nombre__icontains = consulta)
52         serializador = postulanteSerializer(resultado, many=True)
53         return JsonResponse({'ok':True,'data':serializador.data}, status=200)
```

Las líneas 44 a 53 componen el método GET, el cual en primera instancia comprueba el correcto ingreso de datos en el json del request (líneas 44 a 49), detectando en primera instancia si las llaves no corresponden y en segunda si los datos están incorrectos, para las excepciones devuelve el código de error HTTP correspondiente y el mensaje especificando el problema. Finalmente, tras la comprobación de todo lo anterior, las líneas 50 a 53 realizan la consulta a la base de datos y entregan la respuesta satisfactoria del método.

Búsqueda de las 10 mejores opciones de postulación:

```

55 class postular(views.APIView):
56     authentication_classes = [TokenAuthentication]
57     permission_classes = [IsAuthenticated]
58     queryset = postulante.objects.all()
59     serializer_class = postulanteSerializer
60     def post(self, request):
61         ##### SE REVISA EL JSON DE INGRESO
62 >         if ('nem' in request.POST):...
63         else:
64             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'nem'"
65 >         if ('ciencias' in request.POST != True):...
66         else:
67             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'ciencias'"
68 >         if ('historia' in request.POST != True):...
69         else:
70             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'historia'"
71 >         if ('matematicas' in request.POST != True):...
72         else:
73             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'matematicas'"
74 >         if ('lenguaje' in request.POST != True):...
75         else:
76             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'lenguaje'"
77 >         if ('ranking' in request.POST != True):...
78         else:
79             return JsonResponse({'ok':False,'data':"json mal ingresado, falta la llave 'ranking'"
80         ##### SE PREPARA LA RESPUESTA
81         bdddato = postulante.objects.all()
82         opciones = []
83 >         for i in bdddato:...
84         opciones.sort(key=lambda x: x[1], reverse=1)
85         mejoresdiez = opciones[0:10]
86         data = []
87 >         for carrera in mejoresdiez:...
88         return JsonResponse({'ok':True,'data':data}, status=200)

```

Este método presenta ser el más complejo, pero sigue la misma estructura general de los anteriores, en una primera instancia, revisa el json de entrada tanto las keys como sus valores y devuelve las excepciones en caso de haberlas, luego, se preparan los datos de respuesta con el siguiente For:

```

108 for i in bdddato:
109     if (float(json.loads(request.POST['lenguaje']))+float(json.loads(request.POST['matematicas'])))/2<i.PuntajePromedio):
110         return JsonResponse({'ok':True,'data':"Lo sentimos, tu puntaje promedio entre matematicas y lenguaje es inferior a...
111     p0 = (float(i.Nem)/100)*float(json.loads(request.POST['nem']))
112     p1 = (float(i.Ranking)/100)*float(json.loads(request.POST['ranking']))
113     p2 = (float(i.Lenguaje)/100)*float(json.loads(request.POST['lenguaje']))
114     p3 = (float(i.Matematicas)/100)*float(json.loads(request.POST['matematicas']))
115     if (float(json.loads(request.POST['historia']))>float(json.loads(request.POST['ciencias']))):
116         p4 = (float(i.Historia)/100)*float(json.loads(request.POST['historia']))
117     else:
118         p4 = (float(i.Ciencias)/100)*float(json.loads(request.POST['ciencias']))
119     ponderacion = p0+p1+p2+p3+p4
120     posicion = 0
121     contador = 0
122     while(True):
123         puntajexposicion = (i.PrimerMatriculado-i.UltimoMatriculado)/i.Vacantes
124         if (ponderacion>=i.PrimerMatriculado):
125             posicion = 1
126             break
127         if (ponderacion<=i.UltimoMatriculado):
128             posicion = int(i.Vacantes)
129             break
130         if (ponderacion>=i.PrimerMatriculado-(puntajexposicion*contador)):
131             posicion = contador+1
132             break
133         contador = contador+1
134     opciones.append([i.Nombre,ponderacion,i.Codigo,posicion])
135 opciones.sort(key=lambda x: x[1], reverse=1)
136 mejoresdiez = opciones[0:10]
137 data = []

```


Este bucle almacena en “i” cada carrera de la base de datos y su primer objetivo es revisar el puntaje promedio mínimo admitido entre matemáticas y lenguaje (líneas 109 y 110), y si no se cumple entrega la respuesta correspondiente, luego, de las líneas 111 a 114 calcula los puntajes parciales por asignatura del postulante para con la carrera que se esté revisando y en las líneas 115 a 118 comprueba la mejor opción a considerar entre historia o ciencias, finalmente realiza la suma de todos estos valores y se obtiene la ponderación. El siguiente paso es el cálculo de la posición tentativa en la carrera que obtendrá el postulante, para esto se utiliza un bucle para “contar” dicha posición, el cual se ubica en las líneas 122 a 133. Luego de todo lo anterior se tienen los datos de respuesta necesarios por cada una de las carreras, pero, solo son necesarias las 10 mejores considerando solo el puntaje ponderado, como se exige en la solicitud de requerimientos. Para realizar la respuesta de solo las 10 carreras es necesario ordenarlas en base al puntaje ponderado, esta operación se realiza en la línea 135. El código de la línea 135 es muy especial, ya que permite ordenar un “arreglo de arreglos” en base a un dato de los arreglos contenidos, en este caso, dicho dato corresponde al X[1], que no es más que el puntaje ponderado del postulante para cada carrera. Luego de haber realizado el orden de mayor a menor se guardan los primeros 10 resultados en la variable de la línea 136, luego, se crea la estructura json para la respuesta:

```
138     for carrera in mejoresdiez:
139         data.append({
140             'codigo_carrera': carrera[2],
141             'nombre_carrera': carrera[0],
142             'puntaje_postulacion': carrera[1],
143             'lugar_tentativo': carrera[3]})
144     return JsonResponse({'ok':True,'data':data}, status=200)
```

El bucle de las líneas 138 a 143 no es más que la transformación del arreglo normal, a un arreglo de diccionarios el cual en la línea 144 es respondido por el servicio dentro del json.

Información adicional

- Los resultados de puntajes entregados por el servicio consideran decimales, debido a un estudio que se hizo en simuladores de universidades de puntajes psu, donde efectivamente incluyen decimales.
- En el archivo settings.py se pueden ver las aplicaciones y los middleware:

```
57 MIDDLEWARE = [
58     'django.middleware.security.SecurityMiddleware',
59     'django.contrib.sessions.middleware.SessionMiddleware',
60     'django.middleware.common.CommonMiddleware',
61     'django.middleware.csrf.CsrfViewMiddleware',
62     'django.contrib.auth.middleware.AuthenticationMiddleware',
63     'django.contrib.messages.middleware.MessageMiddleware',
64     'django.middleware.clickjacking.XFrameOptionsMiddleware',
65     'corsheaders.middleware.CorsMiddleware',
66 ]
```

```
37  INSTALLED_APPS = [  
38      'django.contrib.sites',  
39      'django.contrib.admin',  
40      'django.contrib.auth',  
41      'django.contrib.contenttypes',  
42      'django.contrib.sessions',  
43      'django.contrib.messages',  
44      'django.contrib.staticfiles',  
45      'puntajes',  
46      'rest_framework',  
47      'corsheaders',  
48      'rest_framework.authtoken',  
49      'rest_auth',  
50      'rest_auth.registration',  
51      'allauth',  
52      'allauth.account',  
53      'allauth.socialaccount',  
54      # 'django_filters',  
55  ]
```

En ellas se puede ver la incorporación del CORS.

Conclusión

Las API REST resultan ser mucho más rápidas de implementar en comparación a una API SOAP, debido principalmente a la alta flexibilidad para el transporte de los datos a diferencia de SOAP, el cual solo permite la comunicación a través de un XML. Otra diferencia importante es que en la API REST los datos pueden ser transportados en una estructura, a diferencia de SOAP que los transporta en un string y sin estructura.

Una vez programada la API fue puesta a prueba con el software Postman, en el cual se probaron los distintos métodos implementados, durante las pruebas nos dimos cuenta que en los servicios que eran llamados mediante el verbo GET, la url iba cambiando en base a las búsquedas realizadas. Esta característica es dada por el método de entrada ya sea Query Params o Path Params y permite ser adaptada por el usuario.

Al utilizar el método POST resultó en que al ingresar los datos en Postman estos no se mostraban en la url, esto debido a que los datos fueron enviados mediante Request Body. La ventaja de este método de entrada es principalmente que los datos se encuentran ocultos para el cliente.

Un aspecto importante dentro del desarrollo fue la elección de un framework, donde finalmente se optó por Django debido a la gran cantidad de funcionalidades que presenta y por contener una estructura de proyecto auto-generado lo que permitió el rápido desarrollo de la aplicación y mantener buenas prácticas durante el desarrollo.