ISPC
INSTITUTO SUPERIOR
POLITÉCNICO CÓRDOBA

TECNICATURA SUPERIOR EN

Desarrollo Web y
Aplicaciones Digitales

NOMBRE DEL ESPACIO CURRICULAR

Módulo

Programador Web

Tema

Primer proyecto en Django

Dirección General de EDUCACIÓN TÉCNICA Y FORMACIÓN PROFESIONAL
Ministerio de EDUCACIÓN
GOBIERNO DE LA PROVINCIA DE CÓRDOBA
entre todos
Hacemos

ÍNDICE

Conceptos básicos

API REST

API REST es un conjunto de reglas y convenciones que permiten que diferentes sistemas y aplicaciones se comuniquen entre sí de manera estandarizada y eficiente. Esta comunicación se basa en el protocolo HTTP, que es el mismo protocolo que se utiliza para navegar por la web.

Una API REST permite que una aplicación exponga ciertos recursos, como datos o funcionalidades, a través de URLs específicas y predefinidas. Estos recursos pueden ser accedidos por otras aplicaciones o sistemas mediante solicitudes HTTP, como GET (para obtener datos), POST (para enviar datos), PUT (para actualizar datos) y DELETE (para eliminar datos).

Además, una API REST sigue ciertas convenciones de diseño, como el uso de recursos autodescriptivos y la separación de la capa de presentación de la capa de datos. Esto facilita el mantenimiento, la escalabilidad y la interoperabilidad de la API.

Es una arquitectura conocida como **cliente-servidor**, en la que el servidor y el cliente actúan de forma independiente, siempre y cuando la interfaz sea la misma al procesar una solicitud y una respuesta, que son los elementos esenciales. **El servidor expone la API REST y el cliente hace uso de ella**. El servidor almacena la información y la pone a disposición del usuario, mientras que el cliente toma la información y la muestra al usuario o la utiliza para realizar posteriores peticiones de más información.

Preparando entorno Django para API REST

Requisito:

- Tener instalado Visual Studio Code
- Tener instalada la extensión de Python para Visual Studio Code
- Tener instalado Python3
- Tener instalado Django

Opcionales:

- Usar una máquina virtual o contenedores.
- Usar virtualenv en caso de no utilizar maquina virtual, esto es para evitar que Django se instale en el entorno “global” de Python.

Opción 2: VirtualEnv

Como primer paso instalamos VirtualEnv con el siguiente comando:

pip install virtualenv

Creamos el entorno

venv mientornovirtual

Luego, activamos el entorno virtual

source mientornovirtual/Scripts/activate

Listo, ya podemos seguir con Django API REST.

Preparando Django

Primero, deben tener instalado Django para poder usar REST.

Para poder programar aplicaciones que usen esta tecnología, tendremos que instalar el paquete, que se hará con el siguiente comando:

pip install djangorestframework

Luego instalaremos el paquete CORS, este paquete nos servirá para poder comunicarnos con Angular cuando lleguemos a ese punto.

pip install django-cors-headers

Continuando nuestro primer proyecto

Con las tablas que nos genera Django crearemos nuestros primeros “EndPoints” que serán “/registro” y “/login”

Para extender el modelo de usuario, vamos a crear un CustomUser en nuestra app. La diferencia más importante respecto a un usuario "clásico" es que, si bien Django maneja como campo identificador el username, nosotros lo identificaremos con el email.

En nuestro archivo models.py:

```python
from django.contrib.auth.models import AbstractUser


class CustomUser(AbstractUser):
    email = models.EmailField(
        max_length=150, unique=True)
```

Según la documentación para establecer el campo identificador único del User tenemos que definir lo siguiente:

```python
USERNAME_FIELD = 'email'
REQUIRED_FIELDS = ['username', 'password']
```

Con estos cambios los usuarios podrán iniciar sesión con el correo en lugar de usar su username.

Hacemos referencia a username y password pero nunca los definimos. Eso es porque se heredan del AbstractUser. De hecho el propio email también se hereda, pero como por defecto no es obligatorio ni único, tenemos que cambiar su configuración tal como hemos hecho.

Ahora vamos a decirle a Django que utilice el CustomUser en lugar de su modelo genérico, para ello iremos al archivo settings.py

```python
# Custom user model
AUTH_USER_MODEL = "authentication.CustomUser"
```

Para poder utilizar el modelo, debemos registrarlo en el archivo admin.py

```python
@admin.register(get_user_model())
class CustomUserAdmin(UserAdmin):
    pass
```

Con esto, ya deberíamos tener acceso a nuestro modelo para crear, editar y borrar usuarios desde el administrador.

Vamos a empezar creando unas vistas básicas de login y logout usando una APIView básica de DRF. La forma de implementar la lógica es exactamente igual que con Django clásico.

Iremos al archivo views.py

```python
from django.contrib.auth import authenticate, login, logout
from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView
```

```python
class LoginView(APIView):
    def post(self, request):
        # Recuperamos las credenciales y autenticamos al usuario
        email = request.data.get('email', None)
        password = request.data.get('password', None)
        user = authenticate(email=email, password=password)

        # Si es correcto añadimos a la request la información de sesión
        if user:
            login(request, user)
            return Response(
                status=status.HTTP_200_OK)

        # Si no es correcto devolvemos un error en la petición
        return Response(
            status=status.HTTP_404_NOT_FOUND)
```

```python
class LogoutView(APIView):
    def post(self, request):
        # Borramos de la request la información de sesión
        logout(request)

        # Devolvemos la respuesta al cliente
        return Response(status=status.HTTP_200_OK)
```

Configuramos las dos URL en la app, en el archivo urls.py de la APP

```python
from django.urls import path, include
from .views import LoginView, LogoutView

urlpatterns = [
    # Auth views
    path('auth/login/',
         LoginView.as_view(), name='auth_login'),

    path('auth/logout/',
         LogoutView.as_view(), name='auth_logout'),
]
```

Y luego en el archivo urls.py pero del proyecto:

```python
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers

# Api router
router = routers.DefaultRouter()

urlpatterns = [
    path('admin/', admin.site.urls),
    # Api routes
    path('api/', include('authentication.urls')),
    path('api/', include(router.urls)),
]
```

Con esta estructura tenemos dos endpoints:

- `/api/auth/login/`
- `/api/auth/logout/`

Probando la autenticación

Ha llegado la hora de probar la API, la forma más fácil es desde la interfaz que nos provee DRF.

Iremos a la URL **login** `http://localhost:8000/api/auth/login/` y escribimos las credenciales como si fuera un objeto JSON:

{ "email": "admin@admin.com", "password": "1234" }

Si todo funciona correctamente al enviar el formulario se verá la respuesta de la API y saldrá nuestro mail arriba a la derecha indicando que efectivamente estamos identificados:

{ "login": "success" }

Serializar

Uno de los requisitos del frontend es que justo después de identificarnos la API debe enviar información básica del usuario para utilizarla en la aplicación, como por ejemplo el nombre, el email o más adelante el avatar.

Cuando nos autenticamos conseguimos un objeto **user** con toda esa información, pero no podemos enviarlo al cliente y ya está, necesitamos transformarlo a un objeto JSON. Ese proceso de transformar el objeto de un formato a otro, Python a Javascript en nuestro caso, se conoce como seralización.

DRF permite crear serializadores de modelos para automatizar esta tarea, así que vamos a crear nuestro propio serializador de usuarios, iremos al archivo serializers.py

```python
from rest_framework import serializers
from django.contrib.auth import get_user_model


class UserSerializer(serializers.ModelSerializer):
    email = serializers.EmailField(
        required=True)
    username = serializers.CharField(
        required=True)
    password = serializers.CharField(
        min_length=8)

    class Meta:
        model = get_user_model()
        fields = ('email', 'username', 'password')
```

Este serializador básico, aparte de controlar los campos que queremos serializar de Python a JSON, también nos servirá más adelante para configurar métodos como la creación de usuarios durante el registro y la validación personalizada de campos.

Sea como sea vamos a serializar el objeto **user** y a enviarlo como respuesta de la petición de login, para ello iremos al archivo views.py

```python
from .serializers import UserSerializer
```

```
# Si es correcto añadimos a la request la información de sesión
if user:
    login(request, user)
    return Response(
        UserSerializer(user).data,
        status=status.HTTP_200_OK)
```

Registro de usuarios

DRF tiene una vista llamada **CreateAPIView** que automatiza la tarea de crear instancias a partir de un serializador, vamos a usarla para facilitarnos la vida:

Iremos al archivo views.py

```
from rest_framework import generics, status
class SignupView(generics.CreateAPIView):
    serializer_class = UserSerializer
```

Luego agregamos esta vista, en el archivo urls.py de la APP:

```
from .views import LoginView, LogoutView, SignupView

urlpatterns = [
    # Auth views
    path('auth/login/',
         LoginView.as_view(), name='auth_login'),

    path('auth/logout/',
         LogoutView.as_view(), name='auth_logout'),
    path('auth/signup/',
         SignupView.as_view(), name='auth_signup'),
]
```

Luego iremos al archivo serializers.py para codificar la contraseña:

```python
from django.contrib.auth.hashers import make_password

def validate_password(self, value):
    return make_password(value)
```

Peticiones CORS

Al acceder a la API desde un cliente web, tal como la tenemos ahora, se nos responderá con este error:

“Access to XMLHttpRequest at 'http://localhost:8000/api/auth/login/' from origin 'http://localhost:3000' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource.”

El error Access-Control-Allow-Origin indica que se ha bloqueado la petición por ser de tipo CORS (Cross Origin Resource Sharing). Esto sucede porque Django no permite peticiones desde distinto hosts y esto es algo que afecta tanto al dominio como al puerto. Al correr Django en el 8000 y el cliente Angular en el 3000, los toma como dos hosts diferentes y salta el error.

En el archivo settings.py, activaremos, en installed apps, la app:

corsheaders

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    "corsheaders",
    'KioscoDeCarlitos',
]
```

La documentación oficial explica que tenemos que configurar un middleware con preferencia, el cuál se encargará de procesar las peticiones CORS:

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    # ...
]
```

Finalmente hay que añadir los dominios que queremos permitir en las peticiones CORS, en nuestro caso el de la URL del cliente:

```
# Configuración de CORS
CORS_ORIGIN_WHITELIST = ["http://localhost:3000"]
```

```
CORS_ALLOW_CREDENTIALS = True
```

Debido a que estamos usando un sistema de autenticación clásico de Django, éste espera que el cliente maneje las credenciales con el atributo withCredentials en las peticiones.

Tampoco debemos olvidar que Django implementa un sistema de seguridad contra exploits CSRF (Cross-Site Request Forgery), por lo que en las peticiones se espera recibir de vuelta una cookie con el csrftoken.