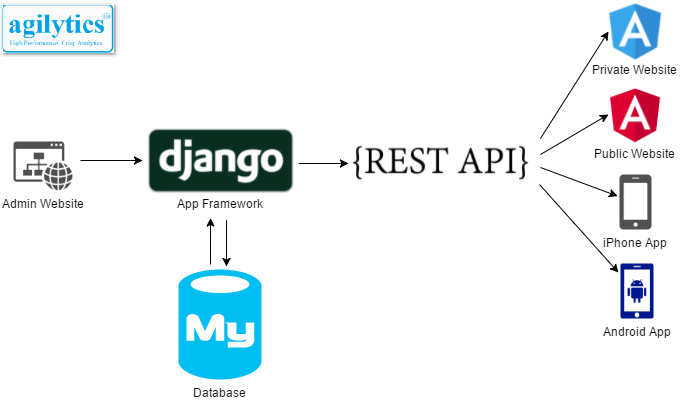
****

*Wykorzystanie API w praktyce*

**Django REST Framework**

**Wstęp**

Tworząc aplikacje z wykorzystaniem Django, nierzadko będziemy potrzebowali, np. zintegrować backend z frontendem aplikacji. Wyobraź sobie program dla kliniki zdrowia, w którym tworzymy m.in kalendarz spotkań. Kalendarz ten ma służyć do zarządzania wizytami pacjentów. Największym problemem jest tutaj fakt, iż takie wizyty są umieszczane w bazie w odpowiednim modelu (załóżmy Appointments), a my takie rekordy musimy wyświetlać i pobierać po stronie JS. Pytanie, jakie się wówczas nasuwa - jak w elegancki sposób umożliwić front-endowej części aplikacji pobieranie danych z bazy?

Odpowiedź jest prosta - za pomocą API zbudowanego z użyciem Django Rest Framework!

**Czym jest DRF?**

**DRF (Django Rest Framework)** to framework, który umożliwia budowania funkcjonalności RESTful API, która pełni rolę pośrednika między widokiem aplikacji a bezpiecznym komunikowaniem się z bazą danych.

DRF dostarcza przede wszystkim **Serializery**, które wykonują zapytania na bazie i służą m.in do **serializowania** otrzymywanych informacji oraz zwracania ich do użytkownika. Aczkolwiek to nie wszystko - serializery mogą nam również posłużyć do **deserializowania** informacji i dodawania rekordów do bazy.

Czym dokładnie jest proces serializacji i deserializacji, poznasz w dalszej części tego szkolenia, dlatego, aby zrozumieć cały proces i złapać lepsze wyobrażenie o tym, jak korzystać z DRF-owych **serializerów i widoków**, zatrzymajmy się na chwilę i poznajmy więcej zagadnień związanych z REST(ful).

**REST API**

**REST - Representational State Transfer** **-** to styl architektury, który określa, w jaki sposób definiowane są zasoby (np. w bazie danych) oraz w jaki sposób uzyskiwać do niej dostęp.

**API - Application Programming Interface -** to zbiór reguł definiujących, w jaki sposób klient może komunikować się z serwerem i uzyskiwać dostęp do określonych zasobów.

Tak więc REST API oznacza całokształt systemu, w którym w określony sposób (przez API) user może manipulować na zasobach (zbudowanych zgodnie ze stylem REST).

**Więcej o REST**

Jak już się zapewne domyślasz, opisywane rozwiązanie jest niezwykle elastyczne i powszechne w użyciu. Przy jego tworzeniu należy mieć jednak szczególną uwagę na kilka zasad, które oceniają, jak bardzo skuteczne jest REST API.   
  
Zdefiniowane one zostały przez Roy’a Fielding’a i brzmią:

* **Client-Server -** w komunikacji powinniśmy bazować na połączeniu Klient-Serwer. Czyli, np. użytkownik korzystający z front-endu sklepu internetowego, wysyła przez API żądania do serwera, który je przetwarza i dodaje zakupiony produkt do koszyka.

Dzięki temu klient jest niezależny od serwera, a serwer niezależny od klienta. Oznacza to, iż klient nie może wpływać na działanie serwera i vice versa. Jedynie mogą się komunikować przez stworzone API.

* **Stateless -** każde żądanie wykonane przez klienta jest niezależne od pozostałych żądań. Serwer nie przechowuje żądań klienta, które mogłyby wpływać na dalsze operacje. Jeżeli będzie potrzeba ich przechowywania, to będą one zapamiętywane po stronie klienta (np. w pamięci cache).
* **Cache -** pamięć podręczna powinna również być wykorzystywana w momencie, gdy chcemy poprawić wydajność aplikacji i, np. nie odwoływać się za każdym razem do bazy danych w celu pobrania pewnych informacji. Przykładem może być sesja i zapamiętywanie w niej pobranego z bazy emailu użytkownika po poprawnym zalogowaniu.
* **Layered system** - wielowarstwowość polega na tym, iż Nasz system powinien działać tak samo wydajnie, niezależnie od tego, czy API dotyczy tylko jednego serwera, czy napotyka po drodze również warstwy pośredniczące, np. load balancer.
* **Code on Demand** - umożliwia przesyłanie od strony serwera pewnych fragmentów kodu, np. JS dla klienta. W praktyce rzadko wykorzystywane.
* **Uniform Interface -** dotyczy tego, iż możemy w różny sposób reprezentować, np. zasoby (API może zwracać pobrane dane z bazy w postaci JSON-a, XML-a czy HTML-a). Jednolitość danych polega też na tym, iż choćby w nagłówku Content-Type protokołu HTTP, możemy odczytywać, w jaki sposób zostały zakodowane dane.

**Więcej o API**

Aby móc lepiej wyobrazić sobie, jak wygląda praca z interfejsem aplikacji, przeanalizujmy cykl komunikacji:

1. Klient preparuje zapytanie w postaci odpowiedniego adresu URI (endpoint), np. *devs-mentoring.pl/students*
2. Klient wysyła przygotowane zapytanie (request),
3. System otrzymuje zapytanie klienta i przygotowuje odpowiedź (response),
4. System zwraca odpowiedź na zapytanie klienta,
5. Klient otrzymuje i przetwarza odpowiedź.

Jednak jest jeszcze jedna rzecz, w zasadzie nieodzowna dla powyższych kroków. Otóż, aby użytkownik mógł skutecznie komunikować się z serwerem - czyli wysyłać żądania i je odbierać, musi przecież korzystać z jakichś zestawu reguł umożliwiających wymianę informacji. Taki zestaw reguł jest definiowany przez powszechny protokół HTTP. Przeanalizujmy jego budowę.

**Budowa zapytania HTTP (request)**

Żądanie w formie HTTP powinno składać się z następujących elementów:

* **Request method** - za jej pomocą określamy, jakie żądania HTTP chcemy wykonać, np. *GET*, *POST*, *PUT*, *DELETE*.
* **Header** - w nagłówku przechowujemy dodatkowe informacje o żądaniu, np. informacje dotyczące plików cookie, kodowania danych.
* **Resource Path** - ścieżka dostępowa do danego zasobu, czyli, tzw. URI
* **Body** - pole opcjonalne. Miejsce do umieszczania dodatkowych informacji, np. danych użytkownika, którego chcemy utworzyć w bazie danych po stronie systemu.

Przykładowy HTTP Request z metodą GET i nagłówkami User-Agent, Host, Accept-Language, Accept-Encoding, Connection, bez pola body:

| GET /devs-mentoring.pl/articles User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT) Host: www.tutorialspoint.com Accept-Language: en-us Accept-Encoding: gzip, deflate Connection: Keep-Alive |
| --- |

**HTTP Request Method:**

Do najbardziej podstawowych metod wykorzystywanych w zapytaniu HTTP należą:

* **GET** - pobieranie danych z serwera
* **POST** - wysyłanie nowych danych do serwera
* **PUT** - podmienianie istniejących danych na serwerze
* **DELETE** - usunięcie danych z serwera
* **PATCH** - częściowa modyfikacja danych na serwerze

Na ten moment nie będę rozpisywał się nad szczegółowym rozpisaniem funkcjonalności wymienionych metod. Dogłębniejsze poznanie ich przyjdzie z czasem, szczególnie wtedy gdy będziemy tworzyli pierwsze aplikacje RESTowe.

**HTTP Header**

Tak już zauważyłeś na powyższych przykładzie, nagłówki zapytania mają postać par **klucz - wartość** i definiują one dodatkowe atrybuty zapytania. Umożliwiają więc one zarówno klientowi, jak i serwerowi na przekazanie dodatkowych informacji. Wyróżniamy wiele rodzajów nagłówków, a ich zastosowanie zależy wyłącznie od kreatywności i potrzeb, np. twórcy aplikacji.   
  
Do najpopularniejszych należy zaliczyć te odpowiadające za uwierzytelnianie, pamięć tymczasową, elementy związane z kodowaniem, bezpieczeństwem itd. Do wyboru do koloru!

**HTTP Resource Path**

URI podawany jest tuż po metodzie HTTP i określa on, do którego zasobu chcemy się odwołać. Pamiętaj, aby podczas projektowania RESTful API, ustalać endpointy (właśnie te wspomniane adresy URI), w taki sposób, aby użyte słowa jasno wyrażały przeznaczenie.

Tak więc w momencie, gdy chcesz pobrać, np. artykuł o ID 1234, pierwszą linię zapytania skonstruujesz tak jak poniżej: **GET devs-mentoring.pl/articles/1234**

**HTTP Message Body**

I na koniec wypadałoby wspomnieć o polu opcjonalnym - ciele zapytania HTTP. Możemy przekazywać w nim dodatkowe wartości, np. dane dla tworzonego rekordu w bazie danych. Przykładowo tak mogłoby wyglądać zapytanie POST z zakodowanymi danymi w postaci JSON-a.

| POST newswebsite.com/articles  Body: {  "article": {  "title": "Example",  "author": "Devs-Mentoring.pl",  "text": "It's working!"  } } |
| --- |

Tak jak widzisz - pole Body często jest wypełniane w momencie, gdy chcemy modyfikować lub dodawać nowe dane do choćby serwerowej bazy danych.

**HTTP Response**

Nie zapominajmy jeszcze o tym, iż oprócz żądań wysyłanych w stronę serwera, otrzymujemy również odpowiedzi, tzw. **responses**, które również bazują na protokole HTTP. Od strony struktury, nie ma dużych różnic między zapytaniem a odpowiedzią.

Jedyna substelna zmiana to brak **HTTP Resource Path** - serwer bowiem zastępuje to pole, tzw. **Response Code**, mówiącym o tym, czy powiodło się dostarczenie zapytania, albo czy serwer funkcjonuje prawidłowo.

I tak na przykład:

Jeżeli wszystko poszło zgodnie z planem, to serwer może odpowiedzieć standardowym kodem **200 (OK)** lub **201 (CREATED)** w przypadku pomyślnego utworzenia nowego rekordu w bazie danych. Jeżeli żądanie klienta zostało błędnie sformułowane (niewłaściwa struktura, szukanie zasobu, który nie istnieje) serwer może odpowiedzieć kodem **400 (BAD REQUEST)** lub **404 (NOT FOUND).**

Wykonując więc choćby takie zapytanie:

| GET devs-mentoring.pl/articles/1234 Accept: application/json |
| --- |

Możemy otrzymać:

| HTTP/1.1 200 (OK) Content-Type: application/json Body:  {  Title: 'Python TOP językiem roku 2020',  Date: 01/01/2021,  Author: Devs-Mentoring.pl } |
| --- |

Jeżeli chodzi o wszystkie rodzaje błędów oraz informacje, z jakim oznaczeniami są one identyfikowane, to odsyłam Cię do dokumentacji, z której sam korzystam na co dzień: [100 Continue - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/100)

**Wracając do DRF...**

Zdaję sobie sprawę, że jeżeli nie miałeś wcześniej do czynienia z budowaniem własnego API, to przedstawione dotąd pojęcia i flow pracy może być niezbyt klarowny. Dlatego przystąpmy do praktyki - zobaczysz wykorzystanie omówionych do tej pory zagadnień w kodzie - i na pewno w ten sposób lepiej zrozumiesz wspomniane wyżej koncepty.

**Problem do rozwiązania**

Kojarzysz stworzoną w poprzednich szkoleniach aplikację bloga? Mieliśmy tam do czynienia z własnoręcznie zdefiniowanymi modelami: **Profile** oraz **Article**. Wyobraź sobie teraz sytuację, że pewna logika biznesowa Twojej aplikacji potrzebuje teraz możliwości dynamicznego przetwarzania informacji związanych z modelem Article. Może to być na przykład chęć dodania pop-upa, który z wykorzystaniem JS będzie dynamicznie wyświetlał preview wszystkich spotkań (zauważ, że tutaj nie sprawdzi się sposób z zaimplementowaniem tego w formie ListView, ponieważ pop-up nie jest oddzielnym widokiem w aplikacji).

Dlatego konieczne będzie wydzielenie w naszym projekcie oddzielnej aplikacji, która będzie pełnić rolę REST API i to z jej funkcjonalności będzie korzystał JS, chcąc pobierać odpowiednie rekordy.

Zacznijmy od podstaw - dodajmy DRF do naszego projektu.

**Instalacja DRF**

Wykonaj w shell:

$ pip install djangorestframework

Zaktualizuj plik **settings.py**:

INSTALLED\_APPS = [

**‘rest\_framework’**,

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

**Utworzenie oddzielnej aplikacji**

Chcąc dodać REST API do naszej aplikacji, utworzymy oddzielną aplikację o nazwie **API**. Dzięki temu będziemy mogli zdefiniować w niej odpowiednie **widoki** i **seralizery**.

Endpointy, na których będziemy bazowali, będą powiązane z widokami, do których będziemy mogli wysyłać odpowiednio spreparowane zapytanie HTTP z przykładowymi metodami: GET, POST, PUT etc.

Nazwy endpointów powinny być pogrupowane w kolekcje i wyglądać mniej więcej w następujący sposób:

|  | **GET** | **POST** | **PUT** | **DELETE** |
| --- | --- | --- | --- | --- |
| /articles/ | Pobierz wszystkie artykuły | Dodaj nowy artykuł | Zaktualizuj wszystkie artykuły | Usuń wszystkie artykuły |
| /article/<id> | Pobierz artykuł o danym id | - | Aktualizuj artykuł o danym id | Usuń artykuł o danym id |

Oczywiście taka postać endpointów nie jest złotą zasadą, równie dobrze moglibyśmy stworzyć oddzielny endpoint dla metody POST, który prezentowałby się: **/article-create/**.

Tak więc do stworzonego dotychczas projektu Blog, możemy dodać aplikację API (dzięki poleceniu: *django-admin startapp API*), odpowiednio ją skonfigurować (pamiętaj o dodaniu odpowiednich zależności do listy settings w aplikacji głównej). Następnie do tak utworzonej aplikacji API, dodamy plik **serializers.py**...

**Stworzenie pierwszego Serializera**

Koncept aplikacji już mamy. Teraz nastąpiła pora na stworzenie pierwszego serializera, który umożliwi nam serializację danych.

Proces serializacji obiektu to nic innego, jak zamiana obiektu na odpowiedni API format, np. JSON lub XML.

Załóżmy więc, że mamy poniższą klasę Article

| class Article(models.Model):  title = models.CharField(max\_length=100)  content = models.TextField()  date\_posted = models.DateTimeField(default=timezone.now)  author = models.ForeignKey(User, on\_delete=models.CASCADE) |
| --- |

oraz utworzony rekord:

*Article(‘Devs-Mentoring’, ‘Django is brilliant!’, author=self.request.user)*

Po przeprowadzeniu procesu serializacji takiego obiektu, otrzymamy następujący JSON:

| {  ‘id’: 1,  ‘title’: ‘Devs-Mentoring’,  ‘content’: ‘Django is brilliant!’,  ‘date\_posted’: “2021-05-02T12:00:56Z”,  ‘author: 7  } |
| --- |

Aby proces takiej serializacji przebiegał maksymalnie automatycznie i wygodnie, wykorzystamy klasę **ModelSerializer**, w której możemy definiować rekordy, jakiej tabeli chcemy zamieniać do JSON-a oraz jakie pola chcemy zawierać w tak stworzonym formacie.

Przejdźmy do wcześniej utworzonego pliku serializers.py i dodajmy do niego następujący kod:

| from rest\_framework import serializers from blog.models import Article  class ArticleSerializer(serializers.ModelSerializer):  class Meta:  model = Article  fields = '\_\_all\_\_' |
| --- |

Stworzona klasa ArticleSerializer jest niczym innym jak strukturą umożliwiającą sprawną serializację obiektów z modelu Article do postaci JSON. Taki format będzie zawierał klucze odzwierciedlające wszystkie kolumny obiektu Article (co określiliśmy w polu **fields**, nadając mu wartość **‘\_\_all\_\_’**; pole to możemy dowolnie modyfikować, np. do postaci (‘id’, ‘title’, ‘content’) - wówczas serializowany będzie obiekt tylko z takimi 3-ema polami).

**Dodanie odpowiednich widoków**

W porządku, serializer już mamy! Teraz nadeszła pora na dodanie odpowiedniego widoku, do którego user będzie wysyłał request (GET), a widok taki zwracał będzie JSONa (który powstanie dzięki wcześniej utworzonemu serializerowi).

Widok ten będzie dostępny pod endpointem **/api/articles/**, gdzie **/api/** to ścieżka do niedawno dodanej aplikacji API (URL ten ustawiliśmy w pliku **urls.py** należącego do głównej aplikacji). Omawiany widok nazwany będzie jako: **GetAllArticles**.

Urls.py w aplikacji głównej:

| from django.urls import path from .views import GetAllArticles  urlpatterns = [  path('articles/', GetAllArticles.as\_view(), name='get\_articles') ] |
| --- |

Urls.py w aplikacji API (na czerwono zaznaczony nowo dodany path):

| urlpatterns = [  path('admin/', admin.site.urls),  path('', include('blog.urls')),  path('register/', user\_views.register, name='register'),  path('login/', auth\_views.LoginView.as\_view(template\_name='users/login.html'), name='login'),  path('logout/', auth\_views.LogoutView.as\_view(template\_name='users/logout.html'), name='logout'),  path('profile/', user\_views.profile, name='profile'),  path('api/', include('API.urls')) ] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT) |
| --- |

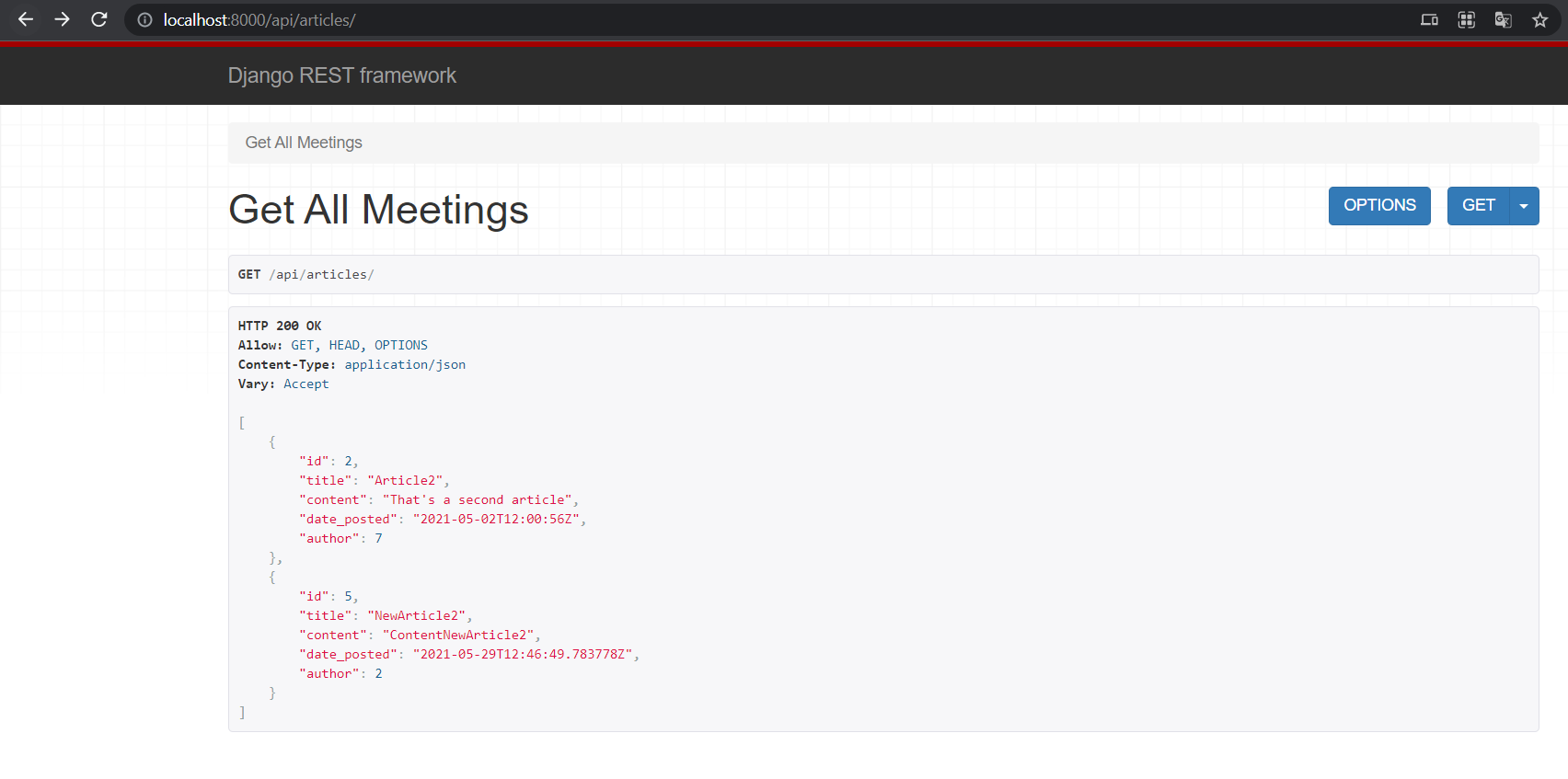
Co do samego widoku (opartego o klasy), to prezentuje się on następująco:

| from rest\_framework.viewsets import generics from .serializers import ArticleSerializer from blog.models import Article  class GetAllArticles(generics.ListAPIView):  serializer\_class = ArticleSerializer  queryset = Article.objects.all() |
| --- |

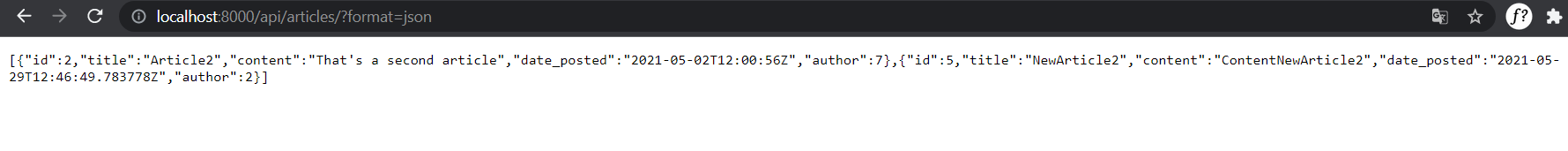
Zwróć uwagę na następujące rzeczy:

1. Klasa GetAllArticles dziedziczy po **generics.ListAPIView**. Taka klasa określa, iż dziedziczący po niej widok może korzystać z metody GET i mieć tylko i wyłącznie dostęp read-only do pobranych elementów.
2. serializer\_class reprezentuje serializer, który będzie przekształcał obiekty z bazy do JSON-a
3. queryset jest natomiast polem niezbędnym do tego, aby określić, co ma być serializowane po wykonaniu zapytania na endpoint-cie reprezentującego GetAllMeetings. W naszym przypadku ma być to lista wszystkich artykułów.

Od teraz, gdy przejdziemy pod adres <http://localhost:8000/api/articles/>, naszym oczom ukaże się następujący widok:



Gdy klikniemy w niebieski przycisk GET i wybierzemy json, możemy zobaczyć w jakiej postaci zostanie zwrócony nam zestaw informacji z bazy.



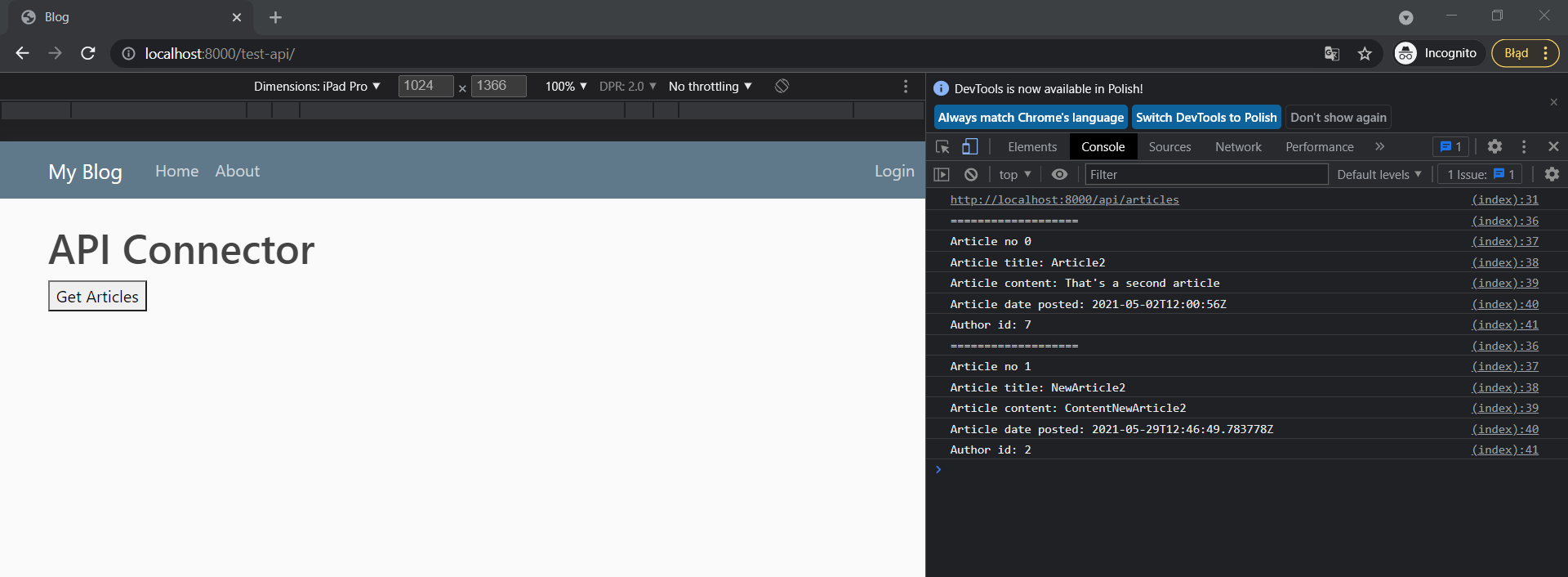
Tak jak wspomniałem wcześniej, bądź świadomy z tego, iż DRF i budowane z jego wykorzystaniem API, nie musi nam służyć tylko i wyłącznie do pobierania danych z bazy w odpowiednim formacie.

Korzystając choćby z takich klas bazowych jak: **CreateApiView**, **DestroyApiView**, **UpdateApiView,** możemy wywoływać endpoint z metodami **POST**, **DELETE**, **PUT/PATCH** i w ten sposób bezpośrednio modyfikować rekordy w bazie. Oczywiście będąc przy operacjach manipulującymi informacjami w bazie, istotną kwestią, która należałoby poruszyć są permission i dodawanie restrykcji - tak, aby nie każdy zalogowany user mógł modyfikować rekordy w bazie. Do tego jednak dojdziemy w dalszej części szkolenia. Teraz natomiast chciałbym przedstawić Ci, jak pobierać dane z utworzonego widoku **GetAllArticles** w JS.

**Wykorzystanie ListApiView w praktyce**

Załóżmy, że chcielibyśmy teraz wyświetlić listę wszystkich spotkań w JavaScriptowej konsoli. Na ten moment ograniczam się do samej konsoli i podstawowego wyglądu aplikacji, ponieważ celem tego szkolenia jest nauczyć Cię, jak stworzyć API i wysyłać odpowiednie requesty z poziomu JS, a nie jak stylować stronę.

W rezultacie, po dodaniu odpowiedniego kodu JS, powinniśmy otrzymać:



Strona test-api działa tak, że po kliknięciu przycisku Get Articles, następuje wysłanie requestu do zbudowanego przez nas API, a serializer ArticleSerializer przekształca listę obiektów zwróconych przez widok GetAllArticles do formatu JSON.

Dzięki temu, w konsoli możemy ujrzeć informację o wszystkich artykułach znajdujących się w bazie. Przeanalizujmy teraz, jak wygląda kod JS umożliwiający odbieranie takich informacji z API.

| {% extends 'blog/base.html' %}  {% block script %} <script> function getBaseUrl(path){  let protocol = window.location.protocol;  let host = window.location.hostname;  let port = window.location.port;   return `${protocol}//${host ? host : ""}:${port}${path}` }  async function getArticles(){  let articlesResp = await fetch(getBaseUrl('/api/articles/'));   let data = await articlesResp.json();  data.forEach((article, index) => {  console.log('===================')  console.log(`Article no ${index}`)  console.log(`Article title: ${article.title}`);  console.log(`Article content: ${article.content}`);  console.log(`Article date posted: ${article.date\_posted}`);  console.log(`Author id: ${article.author}`);  }); } </script> {% endblock %}  {% block content %}  <h1>API Connector</h1>  <**button** onclick="getArticles()">Get Articles</**button**> {% endblock %} |
| --- |

W celu wysyłania requestów do API, użyliśmy JavaScriptowej funkcji **fetch()**. Rozwiązanie to jest związane z Promise-ami i przetwarzaniem asynchronicznym. Zagadnienie to należy do zarządzania front-endem, więc warto go kojarzyć szczególnie, gdy pracujesz nad złożonymi aplikacjami (gdzie konieczne jest utrzymanie i rozwijanie zarówno strony frontowej jak i back-endowej projektu).

**Pozostałe operacje na API**

Super, udało nam się zbudować działające API, które umożliwia nam wykonywanie operacji read-only i odczytywanie JSONa ze wszystkimi obiektami Article z bazy. Pójdźmy krok dalej i sprawdźmy, jaki widok należy stworzyć oraz jak zintegrować go z JS, aby możliwe było pobieranie tylko jednego konkretnego rekordu oraz wykonywanie operacji POST na tabeli.

**Pobranie konkretnego artykułu**

Aby móc wyłuskać za pośrednictwem API tylko jeden konkretny artykuł, użyjemy widoku bazowego **RetrieveApiView**, a klasę po nim dziedziczącą połączymy ze stworzonym już Serializerem.

Plik views.py

| from rest\_framework.viewsets import generics from .serializers import ArticleSerializer from blog.models import Article   class GetAllMeetings(generics.ListAPIView):  serializer\_class = ArticleSerializer  queryset = Article.objects.all()   class GetMeeting(generics.RetrieveAPIView):  serializer\_class = ArticleSerializer  queryset = Article |
| --- |

Plik urls.py

| from django.urls import path from .views import GetAllArticles, GetArticle  urlpatterns = [  path('articles/', GetAllArticles.as\_view(), name='get\_articles'),  path('article/<int:pk>/', GetArticle.as\_view(), name='get\_article') ] |
| --- |

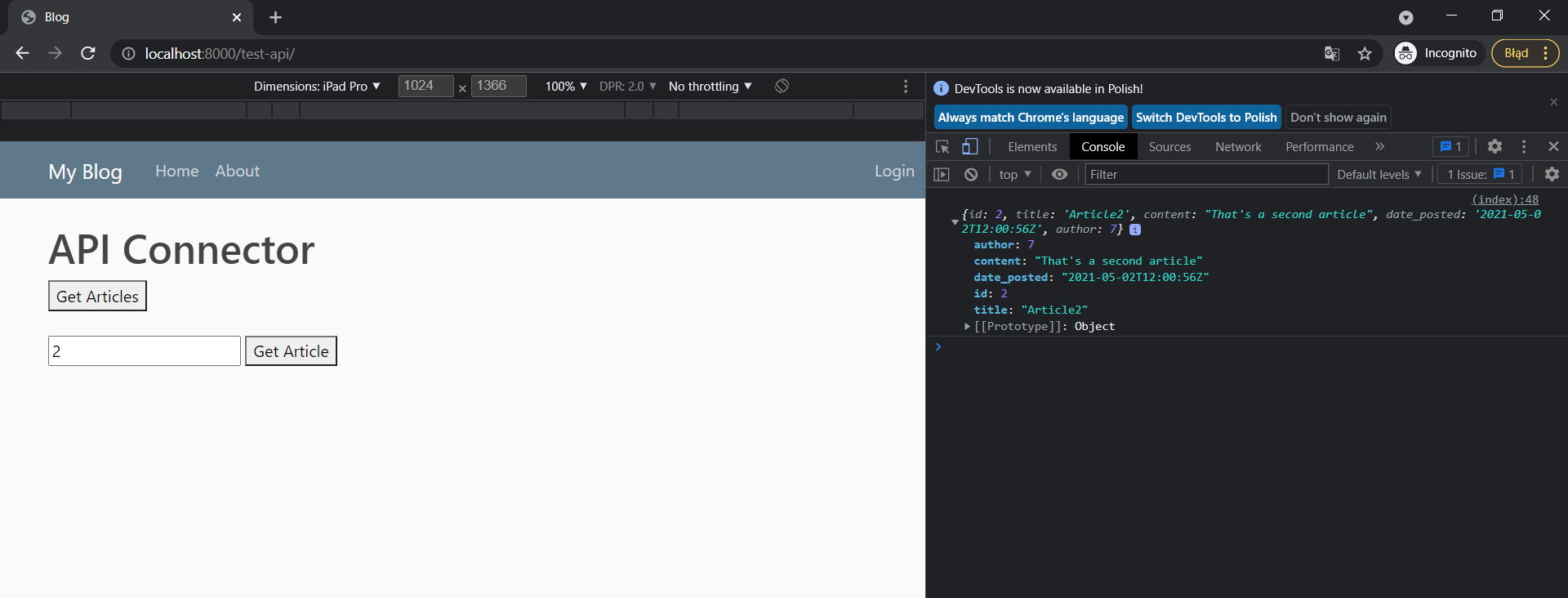
Dużej filozofii tu nie ma - tworzymy class-based view: GetArticle, ustawiamy jej pola **serializer\_class** oraz **queryset** (których zastosowanie omówiliśmy przy tworzeniu poprzedniego widoku - GetArticles).

**Wykorzystanie stworzonego widoku w praktyce**

Rozbudujmy naszą stronę **test-api/** o kolejną prymitywną funkcjonalność, która umożliwi nam pobieranie określonego artykułu przez API.

W dodanej funkcjonalności użyję inputa, do którego będziemy podawali ID artykuły, spod którego chcemy wyciągnąć informacje.

Poniżej podgląd funkcjonalności:



Szablon:

| {% extends 'blog/base.html' %}  {% block script %} <script> function getBaseUrl(path){  let protocol = window.location.protocol;  let host = window.location.hostname;  let port = window.location.port;   return `${protocol}//${host ? host : ""}:${port}${path}` }  async function getArticles(){  let articlesResp = await fetch(getBaseUrl('/api/articles/'));   let data = await articlesResp.json();  data.forEach((article, index) => {  console.log('===================')  console.log(`Article no ${index}`)  console.log(`Article title: ${article.title}`);  console.log(`Article content: ${article.content}`);  console.log(`Article date posted: ${article.date\_posted}`);  console.log(`Author id: ${article.author}`);  }); }  async function getArticle(value){  let articleResp = await fetch(getBaseUrl(`/api/article/${value}`));  let data = await articleResp.json();   console.log(data);  } </script> {% endblock %}  {% block content %}  <h1>API Connector</h1>  <**button** onclick="getArticles()">Get Articles</**button**>  <br/><**br**/>   <input id="article-inputer"/>  <**button** onclick="getArticle(document.getElementById('article-inputer').value)">Get Article</**button**> {% endblock %} |
| --- |

Oczywiście, w powyższym skrypcie brakuje choćby takich aspektów jak walidacja wprowadzonego ID (tak, żeby użytkownik musiał podawać tylko wartość liczbową), czy wyświetlanie odpowiedniego komunikatu, gdy artykuł o podanym ID nie zostanie znaleziony w bazie.nie zostanie znaleziony w bazie. Wprowadzenie takich walidatorów nie jest dla nas priorytetem w tym momencie, a zależy nam głównie na poznaniu DRF i jego możliwości integracji.

**Dodawanie nowego artykułu z DRF**

Aby umożliwić dodawanie nowych artykułów do bazy przez wykorzystanie POST requesta, użyjemy widoku **CreateApiView**. Zarejestrujemy go pod ścieżką **article/create/**.

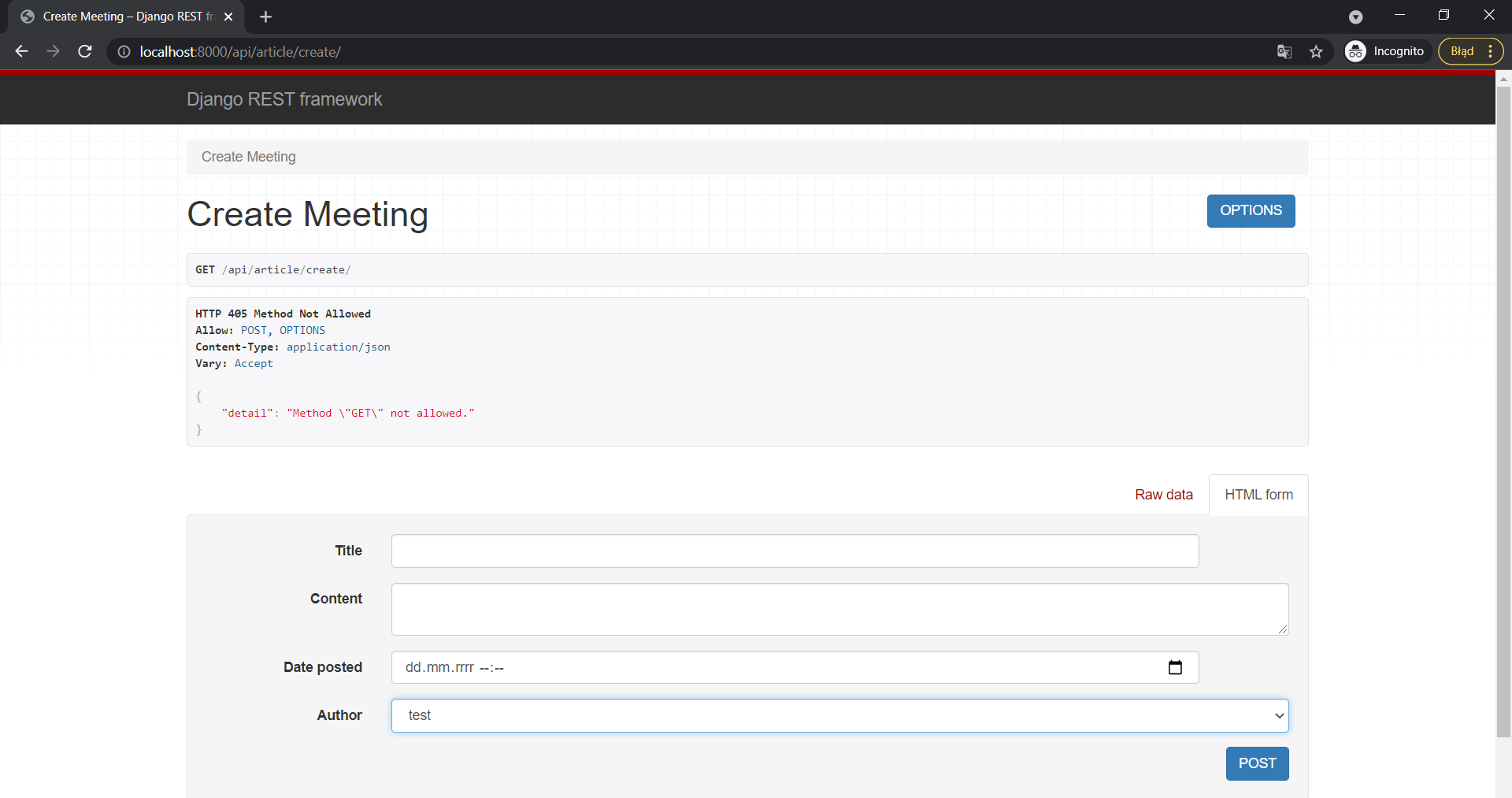
Plik urls.py

| from django.urls import path from .views import GetAllArticles, GetArticle, CreateArticle  urlpatterns = [  path('articles/', GetAllArticles.as\_view(), name='get\_articles'),  path('article/<int:pk>/', GetArticle.as\_view(), name='get\_article'),  path('article/create/', CreateArticle.as\_view(), name='create\_article') ] |
| --- |

Plik views.py

| from rest\_framework.viewsets import generics from .serializers import ArticleSerializer from blog.models import Article   class GetAllArticles(generics.ListAPIView):  serializer\_class = ArticleSerializer  queryset = Article.objects.all()   class GetArticle(generics.RetrieveAPIView):  serializer\_class = ArticleSerializer  queryset = Article   class CreateArticle(generics.CreateAPIView):  serializer\_class = ArticleSerializer |
| --- |

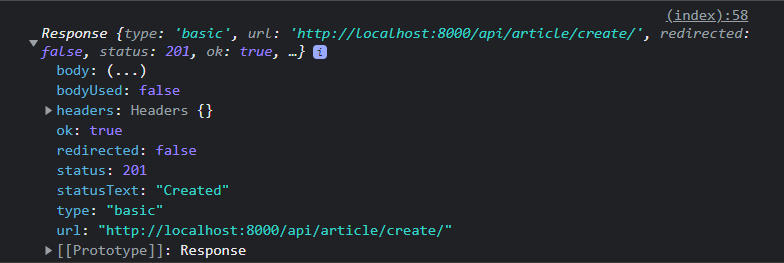
Widok z poziomu localhost:8000/api/article/create



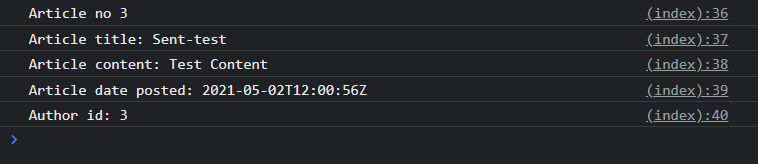
Poniżej standardowo kod JS, który umożliwia tworzenie nowego rekordu w modelu Article (ograniczyłem widok HTML do samej funkcjonalności skryptowej):

| async function addNewArticle(){  let newArticle = {  'id': 6,  'title': 'Sent-test',  'content': 'Test Content',  'author': 3,  'date\_posted': '2021-05-02T12:00:56Z'  }   let addingResp = await fetch(getBaseUrl('/api/article/create/'),  { method: "POST",  headers: { 'Content-Type': 'application/json' },  body: JSON.stringify(newArticle)});  let status = await addingResp;  console.log(status); } |
| --- |

Response:



W związku z tym - gdy klikniemy teraz przycisk Get Articles, ujrzysz na liście artykułów również ten, który został dodany przed chwilą:



Nowy artykuł został poprawnie dodany do bazy!

| **UWAGA**  Wysyłając request do widoku CreateApiView, nie musimy podawać id oraz date\_posted - są to wartości, które mogą być nadawane domyślnie (id jest autoinkrementowalne, a za date\_posted automatycznie wstawiana jest bieżąca data). |
| --- |

**Pozostałe funkcjonalności API**

Oczywiście, nierzadko będziemy chcieli zapewnić również edytowanie bieżących artykułów (metoda PUT lub PATCH), nie wspominając już o możliwości usuwania określonych artykułów. Wówczas użyteczne okaże się wykorzystanie choćby takich metod jak: **DestroyApiView** (usuwanie pojedynczego rekordu), **UpdateApiView** (aktualizacja pojedynczego rekordu), a chcąc zapewnić całą funkcjonalność CRUD, możemy również użyć widoku **RetrieveUpdateDestroyAPIView**, który jak mówi dokumentacja jest: *“Used for read-write-delete endpoints to represent a single model instance. Provides get, put, patch and delete method handlers.”*

Flow pracy z takimi widokami będzie jednak bardzo podobny do tego, wedle którego pracowaliśmy do tej pory - czyli dodanie widoków GetAllArticles, GetArticle oraz CreateArticle. Będziesz miał szansę na przetestowanie takich widoków, wykonując zadania dołączone do tego zestawu materiałów szkoleniowych.

**Często wykorzystywana funkcjonalność w Serializerze**

Zastanówmy się teraz, jak sprawić, aby otrzymywany format daty na frontendzie był przedstawiany w nieco bardziej przyjaznej formie, np. w formacie **DD-MM-YYYY HH:MM:SS**, a nie tak jak jest teraz - zgodnie ze standardem ISO, np. 2021-05-29T12:46:49.783778Z.

Wówczas w klasie ArticleSerializer wystarczy nadpisać metodę to\_representation i odpowiednio pozmieniać w niej format pól, który chcemy przesyłać na front aplikacji:

| class ArticleSerializer(serializers.ModelSerializer):  class Meta:  model = Article  fields = '\_\_all\_\_'   def to\_representation(self, instance):  representation = super().to\_representation(instance)  representation['date\_posted'] = instance.date\_posted.strftime('%d-%m-%Y %H:%M:%S')  return representation |
| --- |

W naszym przypadku zmieniamy tylko pole date\_posted z rekordu Article - tak, aby przykładowo wartość: 2021-05-02T12:00:56Z była zamieniana na 02-05-2021 12:00:56.

**Co dalej?**

Podsumujmy - wiesz już jak w sposób podstawowy korzystać z DRF, jak budować odpowiednie widoki i serializery, z których takie widoki mogą korzystać. W naszej aplikacje brakuje jednak jednego z kluczowych aspektów, o który musimy zadbać, tworząc własne API… Jest to zestaw uprawnień niezbędnych do autoryzowania użytkownika chcącego skorzystać z API!

Zdaj sobie bowiem sprawę, że w tym momencie z naszego API może korzystać dosłownie każdy. Wiąże się to z tym, iż ktoś o złych intencjach może choćby dowolnie dodawać rekordy do bazy.

Aby to uniemożliwić, należałoby dodać co najmniej podstawową weryfikację - czy użytkownik jest zalogowany do naszej aplikacji. Jeżeli nie, to wysyłane przez takiego usera HTTP requesty z metodą POST, byłyby automatycznie odrzucane przez serwer (zakładamy, że artykuły może wyświetlać każdy, tak więc zapytanie GET nie musi być weryfikowane).

Aby zapewnić ograniczony dostęp do widoków API, wykorzystamy wbudowany w rest\_framework podmoduł **permissions**. Jest to biblioteka dostarczająca wiele wbudowanych klas reprezentujących odpowiednie uprawnienia, np. **IsAuthenticated** - sprawdzającą, czy użytkownik próbujący dostać się do danego widoku jest zalogowany, **IsAdminUser** - czy jest adminem etc.

W naszym przypadku idealnie sprawdzi się uprawnienie **IsAuthenticated**, które możemy zaimportować poniższym zapisem:

| from rest\_framework import IsAuthenticated |
| --- |

Teraz, aby połączyć ją z widokiem, wykorzystamy pole **permision\_classes** i dodamy je do CreateArticle.

| class CreateArticle(generics.CreateAPIView):  serializer\_class = ArticleSerializer  permission\_classes = [IsAuthenticated] |
| --- |

Od tego momentu, aby móc wysłać request z metodą POST pod adres /api/article/create (widok CreateArticle), aplikacja będzie przetwarzała requesty tylko od zalogowanych użytkowników. W przeciwnym razie będzie zwracany błąd o error code równym **403 (Forbidden)**.



Pamiętaj jednak, że aby móc przekazywać informacje o tym, czy użytkownik jest zalogowany, czy też nie, musisz do wysyłanego requestu HTTP dodawać nowy nagłówek: 'X-CSRFToken’

Będzie w nim zawarty **CSRF token**, który Djangowy serwer podda analizie i w ten sposób będzie w stanie określić, czy użytkownik jest zalogowany, czy też nie (i zwrócić odpowiedni response).

W związku z tym nasz request, musimy zmodyfikować w następujący sposób:

| let addingResp = await fetch(getBaseUrl('/api/article/create/'), {   method: "POST",  headers: {   'Content-Type': 'application/json',  'X-CSRFToken': getCookie("csrftoken")},  body: JSON.stringify(newArticle) }); |
| --- |

Gdzie **getCookie(“csrftoken”)** to JavaScriptowa funkcja:

| function getCookie(name) {  const value = `; ${document.cookie}`;  const parts = value.split(`; ${name}=`);  if (parts.length === 2) return parts.pop().split(';').shift(); } |
| --- |

Pobiera ona właśnie z cookie potrzebny token unikalnie reprezentującego zalogowanego użytkownika.

**Własne permissions?**

Bądź świadomy również tego, iż jako programiści możemy tworzyć równeiż własne warunki dostępowe do danych widoków. Wystarczy bowiem stworzyć klasę dziedziczącą po, np. **BasePermission** i nadpisać odpowiednie metody, które będą sprawdzały dane warunki.

Przykładowo, możemy zapewnić możliwość dodawania artykułów do bazy tylko tym użytkownikom, którzy są zalogowani oraz są specjalnym typem usera - Redaktorem. Wówczas stworzony permission mógłby wyglądać tak jak poniżej:

| class CreateArticle(BasePermission):  def has\_permission(self, request, view):  if request.user.is\_authenticated:  return request.user.groups.filter(name='Redactor').exists()  return False |
| --- |

Zauważ, że w takiej klasie dziedziczącej po BasePermission konieczne może być override-owanie jednej metody: **has\_permission** oraz **has\_object\_permission**. W pierwszej z nich sprawdzane są uprawnienia read-only do obiektów, a w has\_object\_permission już dostep do konkretnego obiektu (np. konieczność modyfikacji artykułu tylko przez jego autora). Z racji, iż przy tworzeniu nowego artykułu, nie jest potrzebne obiektu w has\_object\_permission, pominąłem użycie tej metody.

Aczkolwiek polecałbym Ci zapoznać się z nimi nieco dokładnie, np. stąd: [Permissions in Django REST Framework](https://testdriven.io/blog/drf-permissions/)

**Zakończenie**

To tyle, jeśli chodzi o podstawy związane z Django Rest Framework. Celem tego szkolenia było zaprezentowanie Ci bazowej funkcjonalności, jaką dostarcza nam ten framework oraz podniesienie świadomości, w jaki sposób korzystać z REST API. Pamiętaj jednak, że zanim będziesz przystępował do pracy z różnymi interfejsami aplikacji, powinieneś zastanowić się, czy nie łatwiejszym i zgrabniejszym sposobem będzie użycie, np. formularzy, które zapewnią automatyczną autoryzację (np. dzięki wykorzystaniu Mixinów) i integrację z modelami w bazie.