TASK

Django – eCommerce Application Part 2

Visit our website

Introduction

WELCOME TO THE DJANGO E-COMMERCE APPLICATION TASK PART 2!

Now that we have built our eCommerce site, a secure web application with authentication and authorisation, it's time to expand our application to support an API. We will be using our knowledge of authentication and authorisation to build a secure web API for our users.

APIS

You might have noticed that it's possible to link certain websites with your social media accounts. Have you ever wondered how this is possible? Web services can choose to build a web API for their service, making it possible for users to add, remove, update, and delete data from the service. Most social media services have a web API that you can integrate with your service. You can also build your own web API that your users can use to integrate with their services.

In this task, we will take a look at how we can create our own web API for our web service, as well as how we can integrate other APIs with our eCommerce service. We will also learn about Postman and how it can be utilised to test your web API.

Web APIs

API is an acronym for "application programming interface" and it's a set of rules that enable software applications to communicate with each other. Web APIs are APIs that allow applications to communicate over the Internet.

Web APIs use standard communication protocols such as HTTP and HTTPS to allow communication over the Internet. Clients can make a request to the API by sending an HTTP request. The API will then process the request and send back a response. This is similar to how our web service already works, but instead of returning HTML, a web API will typically respond with data in XML (Extensible Markup Language) or JSON (JavaScript Object Notation) format. XML defines a set of rules for encoding documents in a format that is both human- and machine-readable, and JSON is an open standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types.

When building a web API, it’s wise to follow some sort of structure or architecture. Using a known standard makes it easier for other developers to use and integrate your API with their projects. It can also assist in preventing issues that can occur later by enabling scalability, flexibility, and independence. We will now take a look at the REST architecture and how we can use it to build a web API for our eCommerce service.

RESTful architecture

In the first task of this two-part task, we asked you to research RESTful APIs. Let’s recap some of what you may have learned. REST (representational state transfer) is a software architecture or guideline, invented by Roy Fielding in 2000, for building APIs. The architecture comprises principles and constraints to achieve better performance, scalability, and modifiability.

The main feature of REST is **statelessness**. This means that each request will be treated as a new request and servers do not save client data between requests. Each request has to contain all the information, such as authentication data, a `GET`/`POST`/`PUT`/`DELETE` command, JSON, XML, etc., needed to understand and process the request. Without all the required information the API will not be able to successfully process the request and will ultimately fail to return the desired resource.

Here are some other features and constraints of REST:

Client-server architecture: REST uses the client-server architecture where the client and server communicate over a uniform interface using HTTP.
Uniform interface: RESTful systems have a uniform and consistent interface, which allows simplicity and scalability. We can create a uniform REST interface by following these four constraints:

- **Identification of resources:** Resources are identified with URIs (Uniform Resource Identifiers).
- **Manipulation of resources through representations:** Clients manipulate resources through the representation provided by the server. Representations should provide all information to modify or delete the resource or fetch additional data.
- **Hypermedia as the engine of application state (HATEOAS):** The client doesn't need prior knowledge of the structure of the application's APIs but can navigate and interact with the application based on the links provided by the server.

Cacheable: When a user makes the same call twice the data is retrieved from the cache instead of making a new API call.
Layered: The client does not know whether it is communicating with an end server or an intermediary such as a gateway.

REPRESENTATIONAL FORMATS

JSON

JSON (JavaScript Object Notation) is a storing and exchanging format that is very popular with web applications. While JSON has "JavaScript" in its name, it's important to note that **the format is language-independent** and only the notation was borrowed from JavaScript.

JSON is human-readable and easy to understand, and it uses key-value pairs similar to a Python dictionary. Values can be of any data type, including string, boolean, integer, float, etc. JSON is also very popular as it can be used in many programming languages, including Python, Java, C#, PHP, and many more.

Here is an example of the JSON format:

```
{
  "name": "Peter Jackson",
  "age": 25,
  "gender": "male",
  "hobbies": ["reading", "gaming", "coding"]
}
```

Take Note

To learn more about JSON you can have a look at the **official website**.

XML

Another format that is also popular to use when communicating data structures is XML (Extensible Markup Language). Just like JSON, XML is platform-independent and human-readable. XML uses tags to mark up elements, and we can create these tags using angle brackets, "<" and ">", as with HTML. Each tag has an opening and closing component, where the closing tag will have a "/" before the element name in the tag (once again, as with HTML).

Here is an example of a basic element:

```xml
<person>
    <name>Peter Jackson</name>
    <age>25</age>
</person>
```

We can add attributes to elements as well to provide additional information:

```xml
<book title="The Catcher in the Rye" author="J.D. Salinger"/>
```

XML documents have a hierarchical structure, which means elements can contain other elements. Here is an example of this:

```xml
<library>
    <book>
        <title>1984</title>
        <author>George Orwell</author>
    </book>
    <book>
        <title>Brave New World</title>
        <author>Aldous Huxley</author>
    </book>
</library>
```

XML's simplicity and flexibility make it suitable for a variety of applications when exchanging and storing information.

MAKING AN API

Let's see how we can build an API for our web app using the RESTful architecture principles.

Basic API request

We can set up a basic API response to see how they are similar to and differ from the regular HTTP responses you might already be familiar with.

```
def basic_api_response(request):
    if request.method == "GET":
        data = serializers.serialize('json', Store.objects.all())
        return JsonResponse(data=data, safe=False)
```

We first make sure that the user has made a `GET` request. We then run the data from our Store models through a serialiser to serialise it to JSON format. Once we have our data in JSON we can return a **`JsonResponse`** instead of an `HttpResponse`. The data from the response is set to the data produced by our serialiser, and we set the `safe` parameter to `False` to make sure any object instance is allowed to be serialised; without this, we can only serialise dictionary objects.

Output:

```
[{\"model\": \"eCommerce.store\", \"pk\": 1, \"fields\": {\"vendor\": 1,
\"name\": \"BaconShop\", \"description\": \"We sell Bacon!\"}},
{\"model\": \"eCommerce.store\", \"pk\": 2, \"fields\": {\"vendor\": 1,
\"name\": \"Sock & Sandals\", \"description\": \"We sell sock! We sell
sandals! Come check them out!\"}}
```

Take Note

Serialisation in Django is where we translate our objects into different formats. To learn more about this, read the **Django documentation**.

Using the Django REST framework

Unfortunately, the response from the method illustrated above does not produce a very user-friendly resource. When people use our API, we want the experience to be easy and efficient. To achieve this, we can use the Django REST framework to make the serialisation of our data a bit easier. You can install the Django Rest framework using the following command:

```
pip install djangorestframework
```

Sometimes you need a class containing a few attributes that would only be useful to one other class. We can call these classes "helper classes". These helper classes also have access to the private attributes of the outer class. To create a helper class, we simply nest a class definition within the class we would like to attach the helper class to.

DRF serializers should be in a separate file, `serializers.py`

In `yourApp/serializers.py`

```
from rest_framework import serializers
From .models import Store
class StoreSerializer(serializers.ModelSerializer):
    class Meta:
        model = Store
        fields = ['vendor', 'name', 'description']
```

Here we created a class called `StoreSerializer` (note that Django uses the American spelling for the command, i.e., a "z" rather than an "s", and so we've reflected this in the class name as well; this is not mandatory, but simply provides consistency with the spelling of the command). `StoreSerializer` will be used to serialise the data from our Store model. We set the superclass to `serializers.ModelSerializer`, then immediately create a new class inside our `StoreSerializer` class called `Meta`. This class will reference the model we are looking to serialise and we can also set the fields that should be added during serialisation.

Now let's update our previous view to use our new serialiser:

In our `yourApp/views.py`

```
from django.http import JsonResponse
from .models import Store
From .serializers import StoreSerializer

def view_stores(request):
    if request.method == "GET":
        stores = Stores.objects.all()

        serializer = StoreSerializer(stores, many=True)
        return JsonResponse(data=serializer.data, safe=False)
```

This time we use `StoreSerializer` instead of `serializers.serialize`. We can then make a query inside our serialiser to get all the values from our store table and serialise them. We also have to state in the serialiser that we are expecting multiple values in the query set by setting the `many` parameter to `True`. Finally, we return a `JsonResponse` object with our dataset to the data produced by our serialiser, and again we set the `safe` parameter to `False` to allow the serialisation of all object instances.

After using our new serialiser, our output looks much better:

```
[{"vendor": 1, "name": "BaconShop", "description": "We sell Bacon!"},
{"vendor": 1, "name": "Sock & Sandals", "description": "We sell sock! We
sell sandals! Come check them out!"}]
```

When creating new views, remember to set their endpoints in the **urls.py** folder of your web app.

```
urlpatterns = [
    path('basic_response/', basic_api_response),
    path('get/stores', view_stores),
]
```

We can use the `api_view` decorator from the Django REST framework to make sure we receive `request` instances in our view, as well as to add context to our `Response` objects to allow content negotiation.

Let’s see how we can add this to our **`view_stores`** view:

```
from rest_framework.decorators import api_view

@api_view(['GET'])
def view_stores(request):
    stores = Store.objects.all()
    serializer = StoreSerializer(stores, many=True)
    return Response(serializer.data)
```

We can also have our response return the data in XML format. But first, we need to install a package to assist us:

```
pip install djangorestframework-xml
```

The **`djangorestframework-xml`** package provides us with serialisers to format our data to XML. First, we have to add the following code to our **settings.py** file:

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework_xml.renderers.XMLRenderer',
    )
}
```

This will add the XML renderer to the rendered classes of our project.

We can then import the **`render_classes`** decorator and apply it to our view using the `XMLRenderer`:

```
from rest_framework.decorators import api_view, renderer_classes
from rest_framework_xml.renderers import XMLRenderer
from rest_framework.response import Response

@api_view(['GET'])
@renderer_classes((XMLRenderer,))
def view_stores(request):
    serializer = StoreSerializer(Store.objects.all(), many=True)
    return Response(serializer.data)
```

The renderer_classes decorator forces this view to render only XML, regardless of any global settings. You will also notice that we return a `Response` object instead of a `JsonResponse` object. The DRF (djangorestframework) will automatically serialise the response using the defined renderer (in this case, XML)

In addition to enabling users to view data using our API, we can allow them to add data, such as creating and editing stores. We can set up the following view for a user to add a store:

```
from rest_framework import status

@api_view(['POST'])
def add_store(request):

    serializer = StoreSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

We first check to see if the request coming in is a `POST` request. We then grab the data from our `request` object and run it through our `StoreSerializer`. Then we test to see if the data from the request is valid and in the correct format. If it's valid, we save the data and return it to the user with a success response code. Otherwise, we return the error messages with a bad request response code. We can return response codes to users by importing the `status` module from `rest_framework`.

When allowing users to view and especially add data to and from our database, we need to perform some authentication. Let's look at how to achieve this:

```
@api_view(['POST'])
@authentication_classes([BasicAuthentication])
@permission_classes([IsAuthenticated])
def add_store(request):
    vendor_id = request.data.get('vendor')
    if vendor_id is None:
        return Response({'error': 'Vendor field is required.'},
status=status.HTTP_400_BAD_REQUEST)
    if int(vendor_id) != request.user.id:
        return Response({'error': 'User ID and vendor ID do not match.'},
status=status.HTTP_403_FORBIDDEN)

    serializer = StoreSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

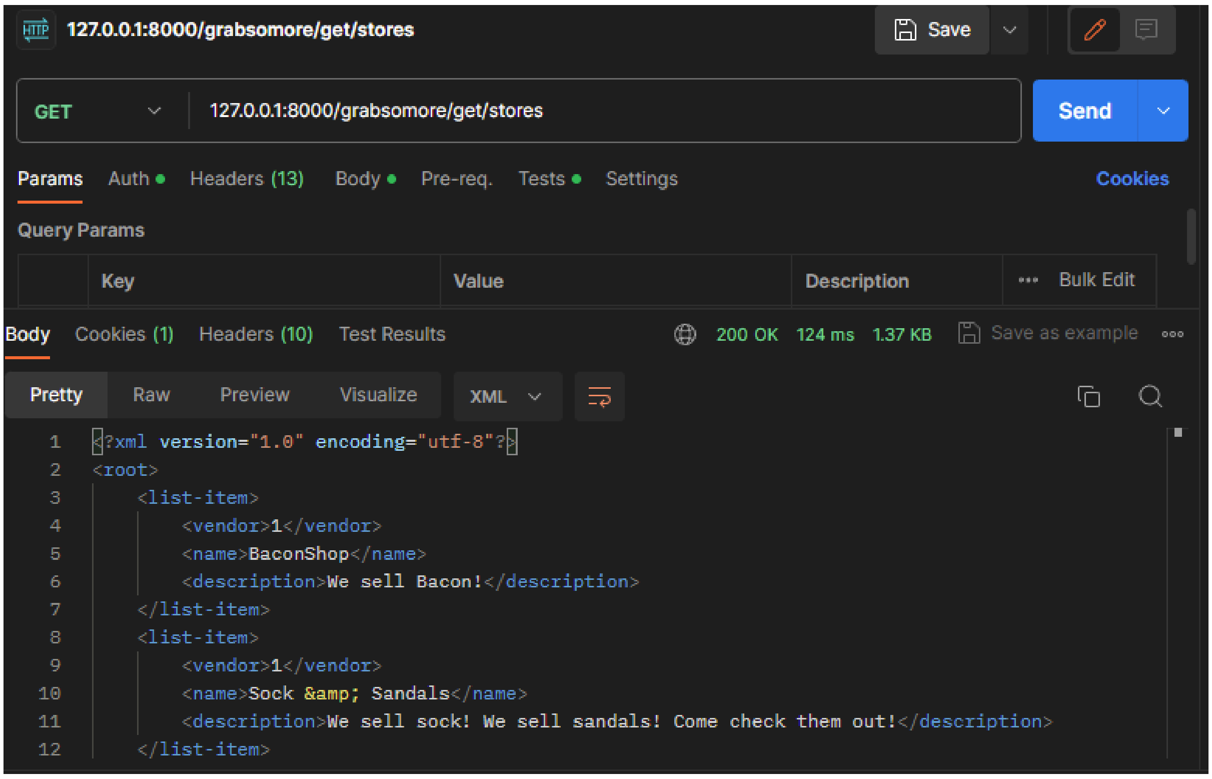
We have now added two more decorators to our function. The first, **`authentication_classes`**, allows us to use a specific authentication method. In this case, we will use **`BasicAuthentication`**, which requires a valid username and password to allow access to the view. We can then determine whether the authentication was successful by using the **`permission_classes`** decorator, adding the **`IsAuthenticated`** permission class to determine whether the user has been authenticated. A bonus of **`BasicAuthentication`** is that it attaches the user to the request. This allows us to determine whether the user is trying to add a new store to their own profile.

POSTMAN

To test our API, it would be quite inefficient to use a browser or to build a new application to make API requests. We can use an application called **Postman** to help us make HTTP requests very easily and attach all the necessary data with it.

In Postman we can add the endpoints to our API where it says "Enter URL or paste text". We can then set the request method and add the required data.

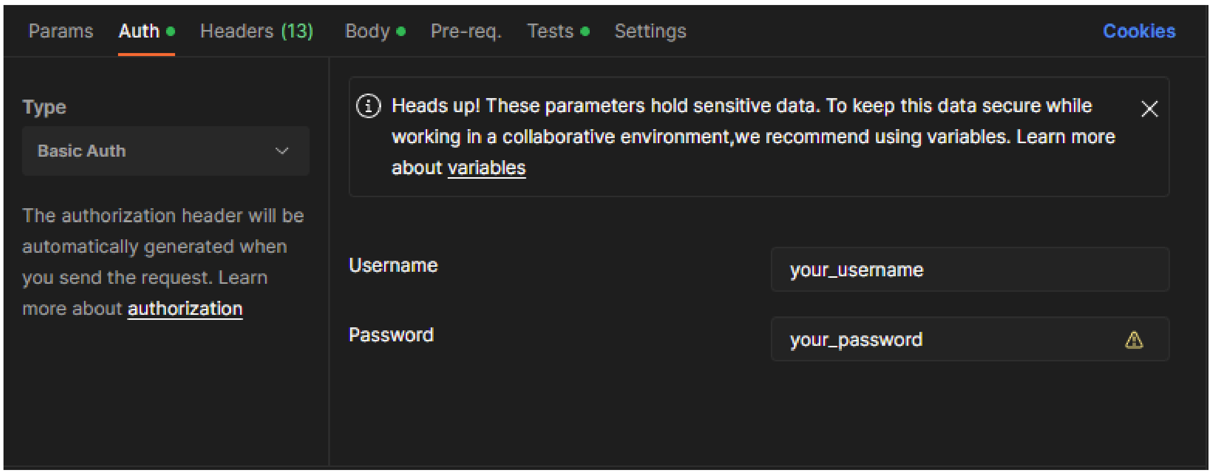
Let's see what Postman will look like if we make a call to our **get/stores** endpoint:



```xml
<?xml version="1.0" encoding="utf-8"?>
<root>
    <list-item>
        <vendor>1</vendor>
        <name>BaconShop</name>
        <description>We sell Bacon!</description>
    </list-item>
    <list-item>
        <vendor>1</vendor>
        <name>Sock &amp; Sandals</name>
        <description>We sell sock! We sell sandals! Come check them out!</description>
    </list-item>
```

We make a request to the **get/stores** endpoint and set our request method to `GET`, as you can see to the left of the endpoint URI. When we hit send, we get the response you can see in the "Body" at the bottom of the image. This time the API returns the resource in XML form.

When making a request to an endpoint that requires authentication, you can navigate to the authentication ("Auth") tab below the URL bar. Once on the authentication tab, you can select the type of authentication. We used "Basic Auth" for our **post/store** endpoint.



Type: Basic Auth

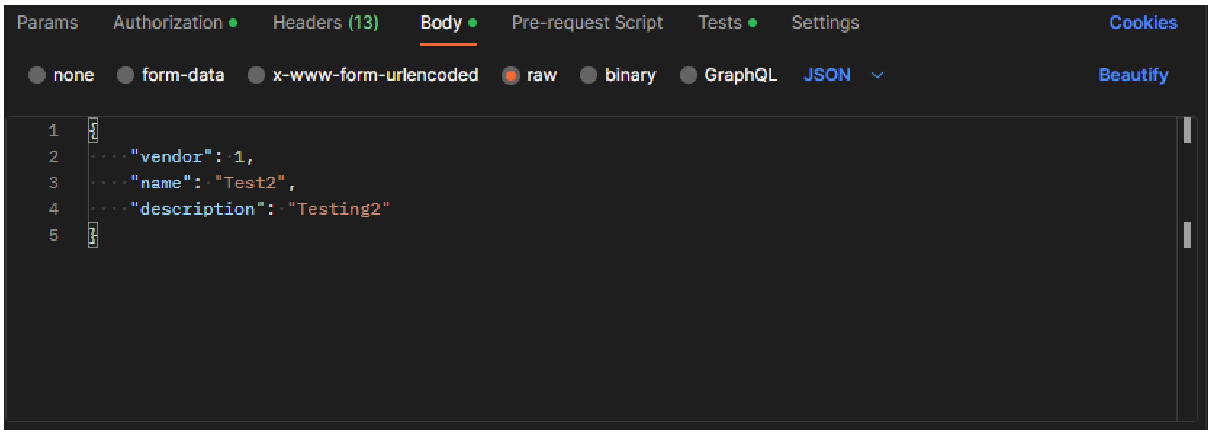
The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables

Username: your_username

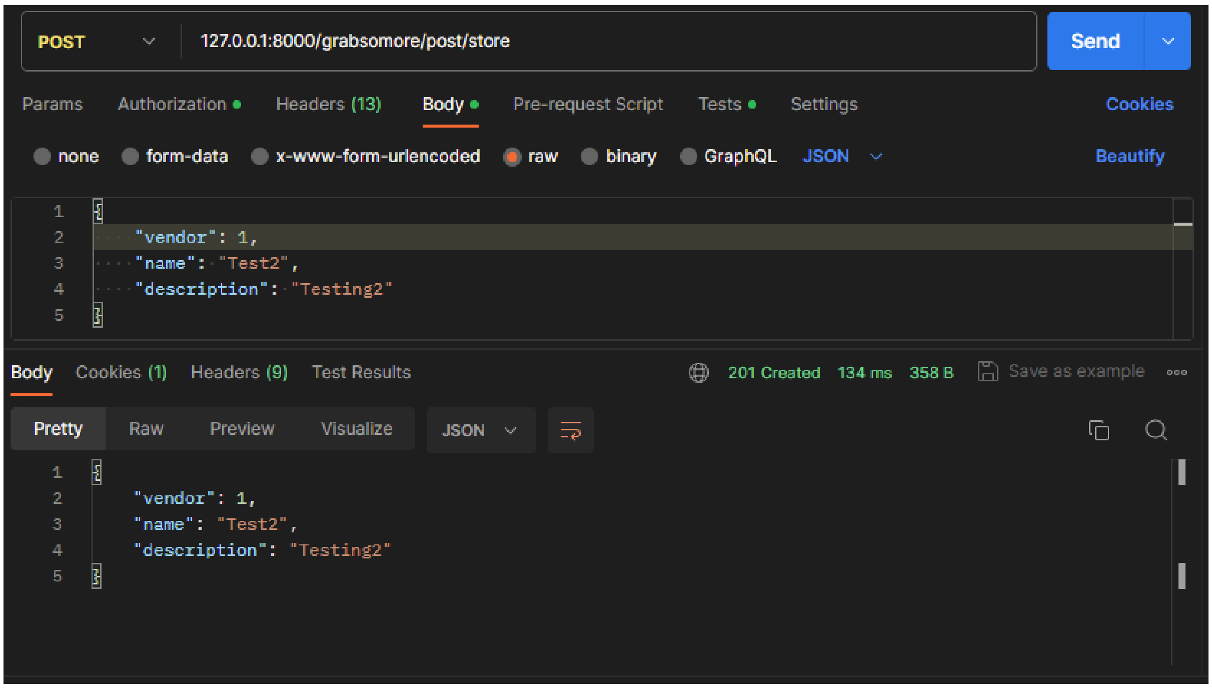
Password: your_password

The **post/store** endpoint also requires the information of a new store in order to add it, and we can add this data to the body of our request. Navigate to the "Body" tab, select the data as "raw", and set the type to "JSON". Then create a store object in JSON format that represents a store in the same way it's produced by our serialiser:



```
{
    "vendor": 1,
    "name": "Test2",
    "description": "Testing2"
}
```

With all our data in place, we can now make a `POST` request to the **post/store** endpoint:



127.0.0.1:8000/grabsomore/post/store

```
{
    "vendor": 1,
    "name": "Test2",
    "description": "Testing2"
}
```

201 Created 134 ms 358 B

```
{
    "vendor": 1,
    "name": "Test2",
    "description": "Testing2"
}
```

INCORPORATING THIRD-PARTY APIS

`Requests` module

Now that we know how an API works it becomes possible for us to incorporate a third-party API into our application. Let's take a look at X (formerly Twitter) and the API they have made available to the public, and consider how we can incorporate that into our own project.

Take Note

Although the name of the company has changed from Twitter to X, the underlying infrastructure is still aligned with the former brand, so the code we will need to write will refer to “Twitter” and “tweets”. References to X and posts made to X in this learning material have thus been aligned to these legacy terms to avoid confusion.

First, we need a way to make API calls easily within Python, and the `requests` module enables us to make our API calls in an elegant and efficient manner. Requests also provide us with an authorisation library, which we will see when using the Twitter API.

Inside your virtual environment, run the following commands in your console:

```
pip install requests
pip install requests-oauthlib
```

Now that we have downloaded the module, we can use it by importing it to our Python file:

```
import requests
```

With `requests` imported, we can simply use the `get()`, `post()`, `put()`, `delete()`, and many more functions available to make our calls. We store the response to the call in a variable as it will contain the resource we have requested.

Here is an example of making a `GET` request:

```
r = requests.get('https://api.github.com/events')
```

If we want to add parameters to our `GET` request, we can do the following:

```
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.get('https://api.github.com/events', params=payload)
```

The request URL will look like this:
`'https://api.github.com/events?key1=value1&key2=value2&key2=value3'`

If we would like to send data with a request, for example a `POST` request, we can add the data to a dictionary and send it with our `POST` as the data parameter:

```
data = {'key': 'value',
      'key2': 'value2'}
r = requests.put('https://httpbin.org/put', data=data)
```

After our request, we can access the content in our response. To get our content in JSON, we can call the `json()` method of `requests`:

```
r = requests.get('https://api.github.com/events')
r.json()
```

From here, we can access the data as you would using a normal Python dictionary.

Take Note

To learn more about `requests` you can have a look at the **official documentation**.

Twitter API

For the Twitter API, we will be using a mixture of `requests` and `requests-oauthlib` to make tweets.

To use the Twitter API, we first need to go to the following **link** to register a developer account. Twitter provides a free tier for their API, allowing you to make up to 1500 calls a month. Once you have registered an account, you will be given an API Key and Secret. Make sure to save these somewhere as you won't be able to view them again and will need to regenerate if you have lost them. Once we have these two keys, we can use them within our program to make some API calls.

We will create our API calls in a separate file; this enables us to simply import the necessary functions within our **views.py** to implement the features. First, we will create a `Tweet` class .

In tweet.py

```
class Tweet():
```

We can then store our API Key and Secret as constants within our class:

```
CONSUMER_KEY = 'your_consumer_key'
CONSUMER_SECRET = 'your_consumer_secret'
```

At the start of our program, we need to authenticate our Key and Secret before we can make any calls. We can set up the following function to achieve this:

```
From requests_oauthlib import OAuth1Session

def authenticate(self):
        # Get request token
        request_token_url =
"https://api.twitter.com/oauth/request_token?oauth_callback=oob&x_auth_acces
s_type=write"
        self.oauth = OAuth1Session(self.CONSUMER_KEY,
client_secret=self.CONSUMER_SECRET)

        try:
            fetch_response =
self.oauth.fetch_request_token(request_token_url)
            print("Request token fetched successfully.")
            return fetch_response
        except ValueError:
            print("Issue with the consumer_key or consumer_secret.")
            return False
```

First, we create a variable to store the URL we will need to request a request token. Next, we create an `OAuth1Session` object using our Key and Secret. With our `OAuth1Session` object, we can call the `fetch_request_token()` method using our request token URL.

From our `fetch` request, we can retrieve the resource owner Key and resource Secret, which we need to create an `OAuth1Session` object that will allow us to make tweets:

```
resource_owner_key = fetch_response.get("oauth_token")
resource_owner_secret = fetch_response.get("oauth_token_secret")
print("Got OAuth token:", resource_owner_key)
```

Take Note

Visit the site to learn more about **OAuth1 and the authentication process**.

Using the base authorisation URL and the `authorization_url()` method, we can generate a URL allowing you to authorise your application. Once authorised, you can copy the PIN provided and paste it into your console to set it as your verifier.

```
# Get authorization
base_authorization_url = "https://api.twitter.com/oauth/authorize"
authorization_url = oauth.authorization_url(base_authorization_url)
print("Please go here and authorize:", authorization_url)
verifier = input("Paste the PIN here: ")
```

Now we can get our access token by again setting the base URL for requesting an access token. Then we create a new `OAuth1Session` object, this time adding our resource owner Key, resource Secret, and our verifier. We then call the `fetch_access_token()` method of our `OAuth1Session` object with our access token URL as the argument.

```
# Get the access token
access_token_url = "https://api.twitter.com/oauth/access_token"
oauth = OAuth1Session(
    self.consumer_key,
    client_secret=self.consumer_secret,
    resource_owner_key=self.resource_owner_key,
    resource_owner_secret=self.resource_owner_secret,
    verifier=verifier
)
try:
    oauth_tokens = oauth.fetch_access_token(access_token_url)
    except ValueError:
        print("Failed to fetch access token.")
        return False
```

Finally, after fetching our access tokens we can retrieve them from our request and create our final **OAuth1Session** object. This time we use the access token and access token secret as our resource owner Key and Secret:

```
access_token = oauth_tokens.get("oauth_token")
access_token_secret = oauth_tokens.get("oauth_token_secret")


# Make the request
self.oauth = OAuth1Session(
            self.consumer_key,
            client_secret=self.consumer_secret,
            resource_owner_key=access_token,
            resource_owner_secret=access_token_secret
        )

        print("Authentication successful. Access token obtained.")
        return True
```

It's important to note that you only have to create this **oauth** object once during the startup of our runtime to authenticate. Creating a new object will require you to go to the authorisation URL again and enter the PIN provided.

Let's take a look at the complete function:

```
def authenticate(self):
        # Get request token
        request_token_url =
"https://api.twitter.com/oauth/request_token?oauth_callback=oob&x_auth_acces
s_type=write"
        oauth = OAuth1Session(self.consumer_key,
client_secret=self.consumer_secret)

        try:
            fetch_response = oauth.fetch_request_token(request_token_url)
        except ValueError:
            print("There may have been an issue with your consumer key or
secret.")
```

```python
            return False

        resource_owner_key = fetch_response.get("oauth_token")
        resource_owner_secret = fetch_response.get("oauth_token_secret")
        print("Got OAuth token:", resource_owner_key)

        # Get user authorization
        base_authorization_url = "https://api.twitter.com/oauth/authorize"
        authorization_url = oauth.authorization_url(base_authorization_url)
        print("Please go here and authorize:", authorization_url)
        verifier = input("Paste the PIN here: ")

        # Exchange verifier for access token
        access_token_url = "https://api.twitter.com/oauth/access_token"
        oauth = OAuth1Session(
            self.consumer_key,
            client_secret=self.consumer_secret,
            resource_owner_key=resource_owner_key,
            resource_owner_secret=resource_owner_secret,
            verifier=verifier
        )
        try:
            oauth_tokens = oauth.fetch_access_token(access_token_url)
        except ValueError:
            print("Failed to fetch access token.")
            return False

        access_token = oauth_tokens.get("oauth_token")
        access_token_secret = oauth_tokens.get("oauth_token_secret")

        # Store authenticated session
        self.oauth = OAuth1Session(
            self.consumer_key,
            client_secret=self.consumer_secret,
            resource_owner_key=access_token,
            resource_owner_secret=access_token_secret
        )

        print("Authentication successful. Access token obtained.")
        return True
```

Now, have a look at the sequence diagram of our interaction with Twitter's API during authentication. As you look at each component, glance back at the code and ensure you can understand which bit of code is doing what.

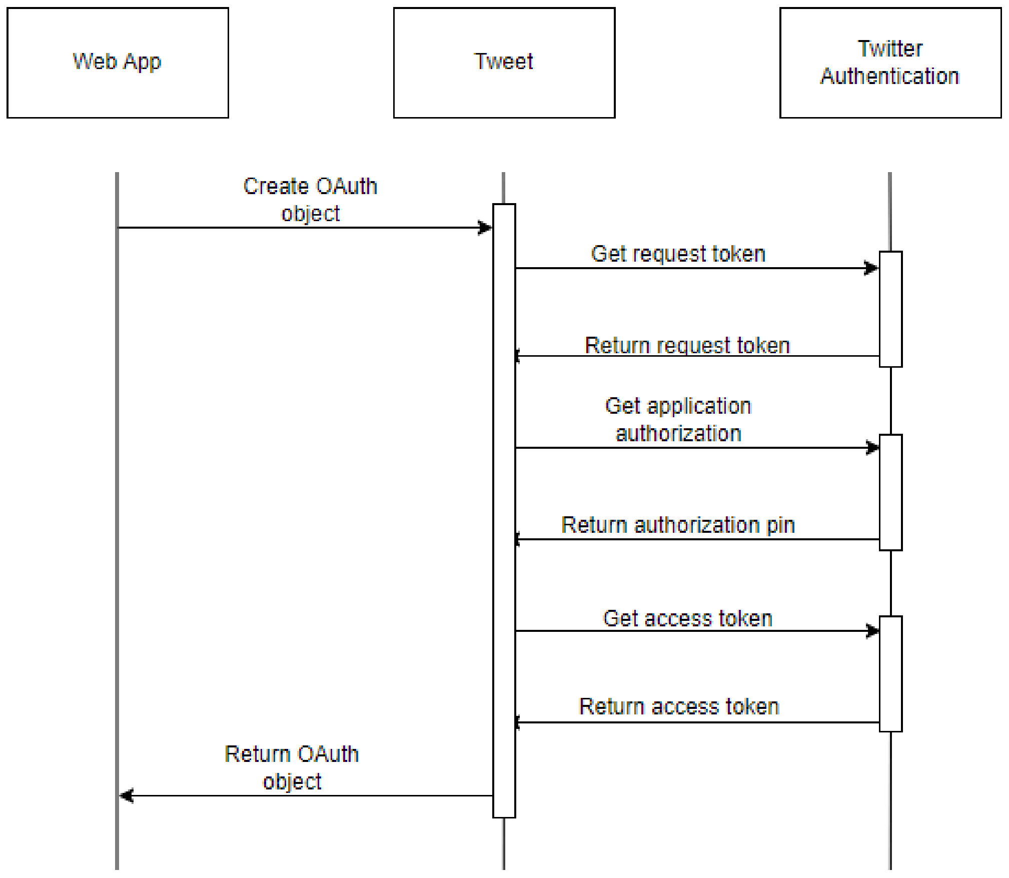


Diagram of our interaction with Twitter's API

Making a tweet

To make a tweet on Twitter, we can create a function with parameters for our `OAuth1Session` object and the tweet.

First, we determine if the app has been authorised. If it has, we can call a `POST` request using `https://api.twitter.com/2/tweets` as our first argument and our tweet as the second.

We have also added a check to determine whether the request has been successful; if not, we throw an exception.

Finally, we retrieve the JSON attached to our response and print it to the console. This JSON response will contain data about the content we have just posted, such as the edit history of the tweet, its ID, and the text of the post.

```python
import json

def make_tweet(self, tweet):
    # Making the request
    if not self.oauth:
        raise ValueError('Authentication failed!')

    response = self.oauth.post(
        "https://api.twitter.com/1.1/statuses/update.json",
        data={"status": tweet},  # Note: not `json=`, but `data=`
    )

    if response.status_code != 200:
        raise Exception(
            f"Tweet failed: {response.status_code} – {response.text}"
        )

    print("Tweet posted successfully!")
    json_response = response.json()
    print(json.dumps(json_response, indent=4, sort_keys=True))
```

The post itself is also in JSON, but since JSON and a Python dictionary work the same, we can create a Python dictionary for our post.

```python
tweet = {"text": body}
```

Take Note

To add more functionality to your post, such as polls and geotags, refer to this **Developer Platform guide**. You can also have a look at all the **sample code** provided by X in the GitHub repository.

When adding the `Tweet` class to our Django project, we want to make sure that only one instance of the object exists. We can implement the **singleton pattern** in our class to ensure a single instance. The singleton pattern is a software design pattern that, when applied to a class, ensures that only a single object of that class type can be created during the program's runtime. Trying to create a new object will just return the original object that has been created. This is similar to creating a global variable within the program that contains our `Tweet` object to use across the program. It's important to note that we only implemented this pattern as we had no other choice and could only allow a single `Tweet` object to exist during our runtime.

To do this, we add the following code:
In tweet.py

```
class Tweet:
    _instance = None  # Singleton instance

    def __new__(cls):
        if cls._instance is None:
            print('Creating the Tweet instance...')
            cls._instance = super(Tweet, cls).__new__(cls)
            cls._instance.oauth = None
            cls._instance.authenticate()
        return cls._instance
```

This code will determine whether a class instance of `Tweet` exists. If it does, it returns the current instance, if it doesn't, a new instance will be created.

Add your Python file containing your `Tweet` class to a directory in your web app directory called **functions**. Inside this **functions** folder add a Python file with the name **__init__.py**. Do not add any code to this file; it is used to enable Python to understand our functions directory as a package.

We can then have the first instance created at the start of our web app runtime. Go to **apps.py** within your web app folder and add the following code:

```
from .funtions.tweet import Tweet
```

Import the `Tweet` class from the **functions** directory and add the following function to the `Config` class:

```
def ready(self):
    Tweet()
```

This will create a first instance allowing us to enter our authentication PIN at the start of our runtime.

Our `Config` class within **apps.py** should now look like this:

```
class EcommerceConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'eCommerce'

    def ready(self):
        Tweet()
```

Now you can import the `Tweet` class to your views and use it to make tweets:

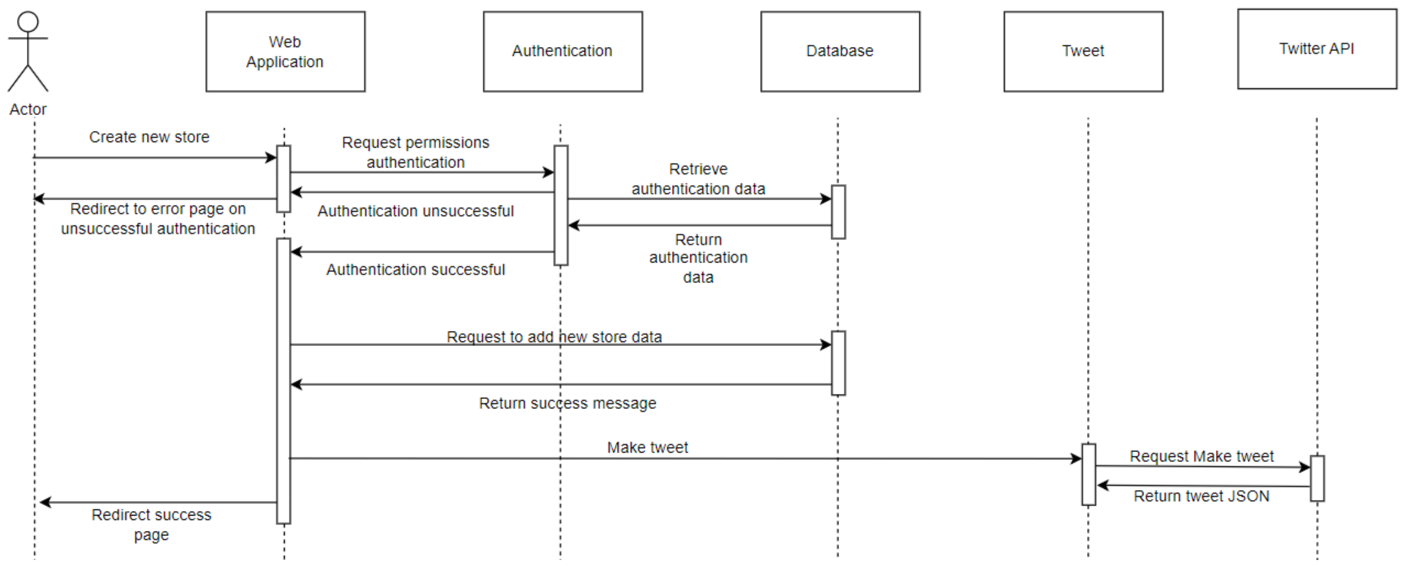
```
from .funtions.tweet import Tweet
```

Here is an example of adding tweet functionality to our `create_store` view to announce a new store on our website:

```
def create_new_store(request):
    if request.method == 'POST' and request.user.is_authenticated:
        user = request.user
        store_name = request.POST.get('store_name', '').strip()
        store_desc = request.POST.get('store_desc', '').strip()

        if not store_name:
            return HttpResponseRedirect(reverse('ecommerce:welcome'))

        new_store = Store.objects.create(
            vendor=user,
            name=store_name,
            description=store_desc
        )

        new_store_tweet = f"New store just launched on Grabsomore!\n\nStore
Name: {new_store.name}\nDescription: {new_store.description}"
        tweet_payload = {'text': new_store_tweet}
        try:
            Tweet._instance.make_tweet(tweet_payload)
        except Exception as e:
            print(f"Error sending tweet: {e}")
        return HttpResponseRedirect(reverse('ecommerce:welcome'))
    return HttpResponseRedirect(reverse('ecommerce:alterlogin'))
```

Here you can see that after creating our new store, we create and send a post about the store before returning an `HttpResponse`.

Let’s take a look at a sequence diagram of our `create_new_store` function:


Actor
Web Application
Authentication
Database
Tweet
Twitter API
Create new store
Request permissions authentication
Retrieve authentication data
Redirect to error page on unsuccessful authentication
Authentication unsuccessful
Return authentication data
Authentication successful
Request to add new store data
Return success message
Make tweet
Request Make tweet
Return tweet JSON
Redirect success page

Sequence diagram of the `create_new_store` function

We can see from the sequence diagram that, when a user creates a new store, the same authentication process from before will happen to ensure the user is allowed to add a store. If authentication is successful a request is sent to the database to add the new store. After the new store is created, the web app sends a request to our **`Tweet`** object to make a tweet about the store. The **`Tweet`** class will then finally request the Twitter API to make a tweet and return the tweet in JSON format. Once this process is complete, the user will be redirected to a page telling them they have successfully added a new store.

Our web app is looking much better now. We have created a RESTful API for it that our users can integrate into their applications or projects. In addition, we have also made use of a third-party API and integrated it into our web app to allow tweets to be posted on X.com about new products and stores.

Practical task 1

In this task, you will extend the functionality of your eCommerce project to allow users to interact with your service via a RESTful API.

Before adding the functionality to your code, you need to plan a few things:

- Think about your models and how you would like to serialise them and have your resources represented (JSON/XML).
- Carefully think about your endpoints as this is how users will access your resources. You can see here for a good set of **best practices**.
- Create a sequence diagram of the interactions a user will have with your API.

Follow these steps:

1. Allow vendors to create new stores and add products to their stores using your web API. They should also be able to retrieve reviews. (Remember to perform authentication before allowing a user to edit resources.)
2. Buyers and vendors should be allowed to retrieve the stores listed under each vendor and the products of each store.

Practical task 2

Now that you have our own API for third-party users, let's incorporate a third-party API into your project.

Follow these steps:

1. You can create a **Twitter (X) developer account** that will allow you to make 1500 free requests each month.

2. Incorporate the Twitter web API into your eCommerce web app to make a tweet when:
    - a new store is added, and
    - a new product is added to a store.
3. A store tweet should contain the store name, description, and logo (if the store has a logo).
4. A product tweet should contain the name of the store that sells the product, the product name, product description, and an image of the product (if available; if there is no image available, ensure the tweet is still sent without the image).

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.