Module 16 - REST APIs using Django

Welcome to Module 16! This module will introduce you to the powerful concept of REST APIs and how to implement them efficiently using Django and the Django REST Framework.

Chapter 1: Introduction to APIs and REST

This chapter lays the groundwork by defining what APIs are and specifically what constitutes a RESTful API.

1.1 APIs and REST APIs

- **What is an API? (Application Programming Interface)**
  - An API is a set of rules and definitions that allows different software applications to communicate with each other.
  - Think of it like a menu in a restaurant: it tells you what you can order (available functions) and how to order it (parameters, data format). You don't need to know how the kitchen prepares the food; you just use the menu.
  - **Examples:** Google Maps API, Twitter API, Payment Gateway APIs.
- **What is REST? (Representational State Transfer)**
  - REST is an architectural style for designing networked applications. It's not a protocol (like HTTP) but a set of guidelines.
  - RESTful APIs are designed to be stateless, client-server, cacheable, and use a uniform interface (HTTP methods).
  - **Key Principles of REST:**
    - **Client-Server:** Separation of concerns. The client (e.g., web browser, mobile app) and server (e.g., your Django API) evolve independently.
    - **Stateless:** Each request from client to server must contain all the information needed to understand the request. The server should not store any client context between requests.
    - **Cacheable:** Responses can be explicitly or implicitly defined as cacheable to improve performance.
    - **Layered System:** A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way.
    - **Uniform Interface:** The most critical constraint. It simplifies system architecture by having a common way to interact with resources. This involves:
      - **Resource-based:** Data is identified as *resources* (e.g., `/products`, `/users/1`).
      - **Standard HTTP Methods:** Use standard HTTP verbs to perform actions on resources:
        - `GET`: Retrieve (Read) a resource or collection of resources. (Idempotent & Safe)
        - `POST`: Create a new resource. (Not Idempotent & Not Safe)

        - `PUT`: Update/Replace an existing resource (entirely). (Idempotent)
        - `PATCH`: Partially update an existing resource. (Not Idempotent)
        - `DELETE`: Remove a resource. (Idempotent)
    - **Stateless Interactions:** No session information is stored on the server.
    - **Hypermedia as the Engine of Application State (HATEOAS):** Resources should contain links to related resources, guiding the client on available actions. (Often not fully implemented in simpler APIs).
  - **Code on Demand (Optional):** Servers can temporarily extend or customize the functionality of a client by transferring executable code.
- **JSON (JavaScript Object Notation):**
  - The most common data format for REST APIs due to its lightweight nature and ease of parsing for both humans and machines.
  - Looks like Python dictionaries and lists.

Chapter 2: Building Your First REST API with DRF

This chapter walks you through setting up Django REST Framework and creating a simple "Hello World" API endpoint.

2.1 Hello World using REST APIs

Let's set up your Django project and create your first API endpoint.

- **1. Create a new Django project and app:**

  Bash

  ```
  django-admin startproject myapi_project
  cd myapi_project
  python manage.py startapp api # Create a new app called 'api'
  ```

- **2. Install Django REST Framework:**

  Bash

  ```
  pip install djangorestframework
  ```

- **3. Add `rest_framework` to `INSTALLED_APPS`:**

  File: `myapi_project/settings.py` **Add:**

  Python

  ```
  # myapi_project/settings.py
  ```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'api', # Your new app
    'rest_framework', # Django REST Framework
]
```

- **4. Create a basic API view (`views.py`):**
    - We'll use DRF's `APIView` class, which allows you to define methods for HTTP verbs (GET, POST, etc.).

File: `api/views.py` **Add:**

Python

```
# api/views.py
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class HelloWorldView(APIView):
    """
    A simple APIView for "Hello World".
    """
    def get(self, request, format=None):
        """
        Returns a static "Hello, REST API World!" message.
        """
        data = {
            "message": "Hello, REST API World!"
        }
        return Response(data, status=status.HTTP_200_OK)
```

Explanation:

- `APIView`: Base class for creating class-based API views.
- `Response`: DRF's `Response` object allows you to return any data that can be rendered into standard content types (like JSON).
- `status`: Provides HTTP status codes for clarity (e.g., `HTTP_200_OK`).

- **5. Define URLs for your API (`urls.py`):**
    - Create a `urls.py` file inside your `api` app.
    - Include this app's URLs in the project's `urls.py`.

File: `api/urls.py` **Add:**

Python

```
# api/urls.py
from django.urls import path
from .views import HelloWorldView
```

```
urlpatterns = [
    path('hello/', HelloWorldView.as_view(), name='hello-world'),
]
```

File: `myapi_project/urls.py` **Add:**

Python

```
# myapi_project/urls.py
from django.contrib import admin
from django.urls import path, include # Import include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')), # Include your API app's URLs
]
```

- **6. Run migrations and the server:**

  Bash

  ```
  python manage.py makemigrations
  python manage.py migrate
  python manage.py runserver
  ```

- **7. Test in your browser:**
  - Open your browser and navigate to `http://127.0.0.1:8000/api/hello/`.
  - **Browser Output:** You should see a browsable API interface provided by DRF, displaying `{"message": "Hello, REST API World!"}`. This browsable API is a great feature for testing and exploring your API.

2.2 Creating model in the admin

Before we create more complex APIs, let's set up a simple Django model to represent our data and make it manageable via the Django admin.

- **1. Define a simple model (`Post`):**
  - This model will represent blog posts.

  File: `api/models.py` **Add:**

  Python

  ```
  # api/models.py
  from django.db import models

  class Post(models.Model):
      title = models.CharField(max_length=200)
      content = models.TextField()
      created_at = models.DateTimeField(auto_now_add=True)
      updated_at = models.DateTimeField(auto_now=True)

      class Meta:
  ```

```
        ordering = ['-created_at'] # Default ordering for posts

    def __str__(self):
        return self.title
```

Explanation:

- `title`: A character field for the post's title.
- `content`: A text field for the post's body.
- `created_at`: Automatically set when the post is first created.
- `updated_at`: Automatically updated every time the post is saved.
- `Meta.ordering`: Specifies the default order for query results (newest posts first).
- `__str__`: Provides a human-readable representation of the object.

- **2. Register the model in Django Admin:**
  - This allows you to create, view, and edit `Post` objects through the Django admin interface.

File: `api/admin.py` **Add:**

Python

```
# api/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

- **3. Create migrations and migrate the database:**

Bash

```
python manage.py makemigrations
python manage.py migrate
```

- **4. Create a superuser (if you haven't already):**
  - You'll need a superuser account to access the admin panel.

Bash

```
python manage.py createsuperuser
# Follow prompts to create username, email, and password
```

- **5. Access Django Admin:**
  - Run the server: `python manage.py runserver`
  - Go to `http://127.0.0.1:8000/admin/` in your browser.
  - Log in with your superuser credentials.
  - You should now see "Posts" under the "API" section. Click on "Posts" to add a few sample posts.

Chapter 3: Creating and Managing API Resources

This chapter focuses on creating APIs to interact with your `Post` model, including posting data.

3.1 Posting APIs (List and Create)

We'll use DRF's `ModelSerializer` and `generics.ListCreateAPIView` to create endpoints for listing all posts and creating new ones.

- **1. Create a `serializers.py` file:**
    - Serializers convert complex data types (like Django model instances) into native Python data types that can then be easily rendered into JSON or XML. They also handle deserialization (parsing incoming data) and validation.

  File: `api/serializers.py` **Add:**

  Python

```
# api/serializers.py
from rest_framework import serializers
from .models import Post

class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = ['id', 'title', 'content', 'created_at',
'updated_at'] # Fields to be included in API output
        read_only_fields = ['created_at', 'updated_at'] # These
fields are not set by client
```

  Explanation:

    - `serializers.ModelSerializer`: A shortcut for creating a serializer that maps directly to a Django model.
    - `Meta.model`: Specifies the model the serializer is for.
    - `Meta.fields`: A list of fields from the model that should be included in the serialized output and accepted for input. `id` is usually included.
- **2. Create List and Create API views (`views.py`):**
    - DRF provides generic views that automate common API patterns. `ListCreateAPIView` handles both `GET` (list all) and `POST` (create new) requests.

  File: `api/views.py` **Modify:**

  Python

```
# api/views.py
# ... (existing imports for HelloWorldView) ...
from rest_framework import generics # Import generics
from .models import Post # Import your Post model
from .serializers import PostSerializer # Import your PostSerializer
```

```
# ... (HelloWorldView class) ...

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all() # The queryset to retrieve objects
    serializer_class = PostSerializer # The serializer to use for
data conversion
```

Explanation:

- `generics.ListCreateAPIView`: A powerful generic view that provides `GET` functionality for listing a queryset and `POST` functionality for creating new instances using the serializer.
- `queryset`: Specifies which objects should be returned when listing.
- `serializer_class`: Specifies which serializer should be used to convert model instances to JSON and vice versa.

- **3. Update URLs (`urls.py`):**

  File: `api/urls.py` **Modify:**

  Python

```
# api/urls.py
from django.urls import path
from .views import HelloWorldView, PostListCreateView # Import
PostListCreateView

urlpatterns = [
    path('hello/', HelloWorldView.as_view(), name='hello-world'),
    path('posts/', PostListCreateView.as_view(), name='post-list-
create'), # New URL for posts
]
```

- **4. Run the server and test:**
  - `python manage.py runserver`
  - **Test GET:** Go to `http://127.0.0.1:8000/api/posts/` in your browser. You should see a list of the posts you created in the admin, formatted by DRF's browsable API.
  - **Test POST (using the browsable API or Postman/Insomnia):**
    - **Browsable API:** On the `http://127.0.0.1:8000/api/posts/` page, scroll down. You'll see a form where you can enter `title` and `content`. Fill them out and click "POST".
    - **Postman/Insomnia:**
      - **Method:** `POST`
      - **URL:** `http://127.0.0.1:8000/api/posts/`
      - **Headers:** `Content-Type: application/json`
      - **Body (raw JSON):**

        JSON

```
{
    "title": "My First API Post",
    "content": "This post was created via the API!"
}
```

- Send the request. You should get a `201 Created` status code and the newly created post's data in the response.

3.2 Segregating into pages (Pagination)

For APIs that return large lists of data, it's crucial to paginate the results to improve performance and user experience.

- **1. Configure default pagination in `settings.py`:**
  - You can set global pagination styles in your DRF settings.

  File: `myapi_project/settings.py` **Add at the end:**

  Python

```
# myapi_project/settings.py

# ... (existing settings) ...

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10 # Number of items per page
}
```

  Explanation:

  - `DEFAULT_PAGINATION_CLASS`: Specifies the pagination class to use globally. `PageNumberPagination` is simple: it uses `page=` and `page_size=` query parameters.
  - `PAGE_SIZE`: Sets the default number of items per page.
- **2. (Optional) Customize pagination per view:**
  - If you need different pagination for specific views, you can override it directly in the view.

  File: `api/views.py` **Modify (Optional):**

  Python

```
# api/views.py
# ... (existing imports) ...
from rest_framework.pagination import PageNumberPagination # Import
pagination class if customizing

# ... (HelloWorldView, PostSerializer) ...

class CustomPagination(PageNumberPagination):
    page_size = 5 # Override global page size for this specific
pagination class
    page_size_query_param = 'page_size' # Allow clients to specify
page_size
    max_page_size = 100 # Maximum page size a client can request

class PostListCreateView(generics.ListCreateAPIView):
```

```
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    pagination_class = CustomPagination # Apply custom pagination to
this view
    # Or simply:
    # pagination_class = PageNumberPagination # Use default DRF
PageNumberPagination directly
```

Explanation: If you set `pagination_class` in a view, it overrides the `DEFAULT_PAGINATION_CLASS` set in `settings.py`.

- **3. Run the server and test:**
  - `python manage.py runserver`
  - **Action:** Add more than 10 posts via the admin or API.
  - **Test:** Go to `http://127.0.0.1:8000/api/posts/`.
  - **Browser Output:** You should now see pagination controls (Previous/Next buttons, page numbers) in the browsable API. The results will be split into pages.
  - **Test with query parameters:**
    - `http://127.0.0.1:8000/api/posts/?page=2` (Go to the second page)
    - `http://127.0.0.1:8000/api/posts/?page=1&page_size=5` (Get 5 items on page 1, if `page_size_query_param` is set)

Chapter 4: Securing and Enhancing Your API

This chapter covers essential security features like authentication and permissions, as well as associating data with users.

4.1 Setting an Authentication

Authentication verifies the identity of a client making a request. DRF provides several authentication schemes. We'll use `TokenAuthentication` and `SessionAuthentication`.

- **1. Add `rest_framework.authtoken` to `INSTALLED_APPS`:**

  File: `myapi_project/settings.py` **Modify:**

  Python

```
# myapi_project/settings.py

INSTALLED_APPS = [
    # ... (existing apps) ...
    'rest_framework',
    'rest_framework.authtoken', # Add this for Token Authentication
]
```

- **2. Run migrations:**

  Bash

  ```
  python manage.py migrate
  ```

  Explanation: This will create a `Token` model in your database, which stores unique tokens for each user.

- **3. Configure default authentication classes in `settings.py`:**

  File: `myapi_project/settings.py` **Modify `REST_FRAMEWORK` dictionary:**

  Python

  ```
  # myapi_project/settings.py

  REST_FRAMEWORK = {
      'DEFAULT_PAGINATION_CLASS':
  'rest_framework.pagination.PageNumberPagination',
      'PAGE_SIZE': 10,
      'DEFAULT_AUTHENTICATION_CLASSES': [ # Add authentication classes
          'rest_framework.authentication.SessionAuthentication', # For
  browsable API and Django sessions
          'rest_framework.authentication.TokenAuthentication',   # For
  programmatic clients (e.g., mobile apps, Postman)
      ],
  }
  ```

  Explanation:

  - `SessionAuthentication`: Allows users authenticated via Django's session (e.g., logged into the admin) to interact with the API via the browsable API.
  - `TokenAuthentication`: Requires a unique token (generated for each user) to be sent in the `Authorization` header for API requests.

- **4. Get/Create a user token:**
  - Tokens are generated per user. You can create one manually for a user.

  Bash

  ```
  python manage.py createsuperuser # If you haven't done so already
  python manage.py drf_create_token <username> # Replace <username>
  with an actual username
  ```

  Example Output:

  ```
  Generated token 9944a9561633cf483984d7285514f77759d5718a for user
  Alice.
  ```

  Explanation: This command creates a token for the specified user and displays it.

- **5. Test API with authentication:**
  - **Via Browsable API:** Go to `http://127.0.0.1:8000/api/posts/`. You should now see a login link in the top right. Log in with your superuser credentials. After logging in, you'll be able to make POST requests via the form.
  - **Via Postman/Insomnia (Token Authentication):**
    - **Method:** `POST` (or GET)
    - **URL:** `http://127.0.0.1:8000/api/posts/`
    - **Headers:**
      - `Content-Type: application/json`
      - `Authorization: Token <your_generated_token_here>` (e.g., `Authorization: Token 9944a9561633cf483984d7285514f77759d5718a`)
    - **Body (raw JSON for POST):**

      JSON

      ```
      {
          "title": "Authenticated Post",
          "content": "This post was made by an authenticated
      user!"
      }
      ```

    - Send the request. If the token is valid, it should succeed. If no token or invalid, you'll get a `401 Unauthorized` or `403 Forbidden` error.

4.2 Adding username to the posts

It's common for API resources to be associated with the user who created them.

- **1. Modify the `Post` model to include an `owner` field:**

  File: `api/models.py` **Modify:**

  Python

  ```
  # api/models.py
  from django.db import models
  from django.contrib.auth.models import User # Import Django's built-
  in User model

  class Post(models.Model):
      owner = models.ForeignKey(User, related_name='posts',
  on_delete=models.CASCADE) # Link to User
      title = models.CharField(max_length=200)
      content = models.TextField()
      created_at = models.DateTimeField(auto_now_add=True)
      updated_at = models.DateTimeField(auto_now=True)

      class Meta:
          ordering = ['-created_at']

      def __str__(self):
          return f"{self.title} by {self.owner.username}"
  ```

Explanation:

- `owner = models.ForeignKey(User, ...)`: Creates a one-to-many relationship where one `User` can own many `Post` objects.
- `related_name='posts'`: Allows you to access related posts from a `User` object (e.g., `user.posts.all()`).
- `on_delete=models.CASCADE`: If a `User` is deleted, all their associated `Post` objects will also be deleted.

- **2. Make migrations and migrate:**

  Bash

  ```
  python manage.py makemigrations
  python manage.py migrate
  ```

  Note: You might be prompted to provide a default for existing posts. Choose option 1 and enter an existing superuser's ID (e.g., `1`) to associate existing posts with that user.

- **3. Update `PostSerializer`:**
  - We need to include the `owner` in the serialized output. Make it `read_only` because clients shouldn't set the owner directly.

  File: `api/serializers.py` **Modify:**

  Python

  ```
  # api/serializers.py
  from rest_framework import serializers
  from .models import Post
  from django.contrib.auth.models import User # Import User model

  class PostSerializer(serializers.ModelSerializer):
      owner = serializers.ReadOnlyField(source='owner.username') #
  Display username as read-only field

      class Meta:
          model = Post
          fields = ['id', 'owner', 'title', 'content', 'created_at',
  'updated_at'] # Include 'owner'
          read_only_fields = ['created_at', 'updated_at']
  ```

  Explanation:

  - `owner = serializers.ReadOnlyField(source='owner.username')`: This makes the `owner` field read-only and tells the serializer to get the value from the `username` attribute of the `owner` object (e.g., "Alice" instead of user ID 1).

- **4. Override `perform_create` in `PostListCreateView` (`views.py`):**
  - When a new post is created via the API, we need to automatically set its `owner` to the currently authenticated user.

  File: `api/views.py` **Modify:**

Python

```
# api/views.py
# ... (existing imports) ...

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    def perform_create(self, serializer):
        """
        Assigns the owner of the post to the authenticated user.
        """
        serializer.save(owner=self.request.user) # Set owner to the
current authenticated user
```

Explanation:

- `perform_create(self, serializer)`: This method is called by DRF's generic views when a new object is being created.
- `serializer.save(owner=self.request.user)`: Instead of just `serializer.save()`, we explicitly pass the `owner` argument, setting it to `self.request.user`, which is the `User` object of the authenticated client.

- **5. Run the server and test:**
  - `python manage.py runserver`
  - **Action:** Log in via the browsable API or use a token in Postman.
  - **Test POST:** Create a new post.
  - **Browser/Postman Output:** The response will now include an `owner` field with the username of the user who made the request.
  - **Test GET:** View the list of posts. Each post should now display its `owner`.

4.3 Adding Permissions

Permissions determine whether a request (from an authenticated user or not) should be granted access to a particular object or view.

- **1. Configure default permission classes in `settings.py`:**
  - This sets a global default for all API views.

File: `myapi_project/settings.py` **Modify `REST_FRAMEWORK` dictionary:**

Python

```
# myapi_project/settings.py

REST_FRAMEWORK = {
    # ... (existing settings) ...
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticatedOrReadOnly', #
Allow authenticated users full access, others read-only
    ],
    # ... (other settings) ...
}
```

Explanation:

- `IsAuthenticatedOrReadOnly`: This is a very common permission.
  - Authenticated users (`IsAuthenticated`) have full read/write access.
  - Unauthenticated users (`ReadOnly`) only have read access (GET, HEAD, OPTIONS).
  - This means anonymous users can view posts but cannot create, update, or delete them.

- **2. (Optional) Customize permissions per view:**
  - You can set `permission_classes` directly in a view to override the global setting.

File: `api/views.py` **Modify `PostListCreateView`:**

Python

```
# api/views.py
# ... (existing imports) ...
from rest_framework import permissions # Import permissions

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly] #
Explicitly set, but same as default here
    # For example, if you wanted only admins to create posts:
    # permission_classes = [permissions.IsAdminUser]

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

Common DRF Permission Classes:

- `AllowAny`: No restrictions.
- `IsAuthenticated`: Only authenticated users can access.
- `IsAdminUser`: Only users with `is_staff=True` (admin users) can access.
- `IsAuthenticatedOrReadOnly`: Authenticated users have full access; unauthenticated users have read-only access.

- **3. Create a custom permission (`IsOwnerOrReadOnly`):**
  - For individual post details, you often want only the owner to be able to update/delete their own post, while others can only view it.

File: `api/permissions.py` **Add new file:**

Python

```
# api/permissions.py
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Custom permission to allow only owners of an object to edit or
delete it.
    Read-only access is allowed for any authenticated user.
```

```
    """
    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request,
        # so we'll always allow GET, HEAD, or OPTIONS requests.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Write permissions are only allowed to the owner of the
snippet.
        return obj.owner == request.user
```

Explanation:

- `permissions.BasePermission`: The base class for custom permissions.
- `has_object_permission(self, request, view, obj)`: This method is called for *object-level* permissions (i.e., when accessing a specific instance like `/posts/1/`).
- `permissions.SAFE_METHODS`: Includes `GET`, `HEAD`, `OPTIONS`. These are considered "safe" (don't modify data).
- `obj.owner == request.user`: This is the core logic: allow write access only if the `owner` of the `obj` (the post) is the same as the `request.user` (the authenticated user).

- **4. Add a `PostDetailView` and apply `IsOwnerOrReadOnly` permission:**

  File: `api/views.py` **Add new view:**

  Python

```
# api/views.py
# ... (existing imports) ...
from .permissions import IsOwnerOrReadOnly # Import your custom
permission

# ... (HelloWorldView, PostListCreateView) ...

class PostDetailView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [IsOwnerOrReadOnly] # Apply custom
permission
```

  Explanation:

  - `generics.RetrieveUpdateDestroyAPIView`: Provides `GET` (retrieve single), `PUT` (full update), `PATCH` (partial update), and `DELETE` (delete) functionality for a single model instance.

- **5. Update URLs for `PostDetailView`:**

  File: `api/urls.py` **Modify:**

  Python

```
# api/urls.py
from django.urls import path
```

```
from .views import HelloWorldView, PostListCreateView, PostDetailView
# Import PostDetailView

urlpatterns = [
    path('hello/', HelloWorldView.as_view(), name='hello-world'),
    path('posts/', PostListCreateView.as_view(), name='post-list-
create'),
    path('posts/<int:pk>/', PostDetailView.as_view(), name='post-
detail'), # New URL for single post
]
```

Explanation: `pk` (primary key) is the standard argument name used by DRF's generic views to identify a specific object.

- **6. Run the server and test permissions:**
  - `python manage.py runserver`
  - **Test unauthenticated GET:** `http://127.0.0.1:8000/api/posts/1/` (should work)
  - **Test unauthenticated PUT/PATCH/DELETE:** Try editing/deleting a post via the browsable API form or Postman without logging in/providing a token. You should get `403 Forbidden`.
  - **Test authenticated PUT/PATCH/DELETE (not owner):**
    - Create two users (e.g., Alice, Bob).
    - Create a post with Alice's token.
    - Log in as Bob (or use Bob's token). Try to update/delete Alice's post (`/posts/<alice_post_id>/`). You should get `403 Forbidden` because Bob is not the owner.
  - **Test authenticated PUT/PATCH/DELETE (owner):**
    - Log in as Alice (or use Alice's token).
    - Update/delete Alice's own post. It should succeed (`200 OK` or `204 No Content`).

4.4 Filtering based on usernames

We can allow clients to filter the list of posts based on the owner's username.

- **1. Install `django-filter`:**

  Bash

  ```
  pip install django-filter
  ```

- **2. Add `django_filters` to `INSTALLED_APPS`:**

  File: `myapi_project/settings.py` **Modify:**

  Python

  ```
  # myapi_project/settings.py

  INSTALLED_APPS = [
      # ... (existing apps) ...
      'rest_framework',
  ```

```
    'rest_framework.authtoken',
    'django_filters', # Add django_filters
]
```

- **3. Configure `DjangoFilterBackend` in `settings.py` (optional, but good default):**

  File: `myapi_project/settings.py` **Modify `REST_FRAMEWORK` dictionary:**

  Python

```
# myapi_project/settings.py

REST_FRAMEWORK = {
    # ... (existing settings) ...
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend', # Add
this filter backend
    ],
    # ... (other settings) ...
}
```

- **4. Specify `filterset_fields` in `PostListCreateView`:**

  File: `api/views.py` **Modify:**

  Python

```
# api/views.py
# ... (existing imports) ...
from django_filters.rest_framework import DjangoFilterBackend #
Import if you set per-view

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]

    # Add filter backends and fields for this view
    filter_backends = [DjangoFilterBackend] # Specify filter backend
for this view
    filterset_fields = ['owner__username', 'title'] # Filter by
owner's username and title
```

  Explanation:

  - `filter_backends = [DjangoFilterBackend]`: Tells this view to use the Django Filter backend.
  - `filterset_fields = ['owner__username', 'title']`: Specifies that clients can filter posts by the `username` field of the `owner` relationship (using Django's __ lookup syntax) and by `title`.
- **5. Run the server and test:**
  - `python manage.py runserver`
  - **Action:** Create posts from different users (e.g., Alice, Bob).
  - **Test:** Go to `http://127.0.0.1:8000/api/posts/`. In the browsable API, you'll see a "Filters" button.

- **Test with query parameters:**
  - `http://127.0.0.1:8000/api/posts/?owner__username=Alice` (Show only posts by Alice)
  - `http://127.0.0.1:8000/api/posts/?title=first` (Show posts with 'first' in the title)

4.5 Search Filter

The search filter allows clients to perform simple searches across specified fields using a single query parameter.

- **1. Add `rest_framework.filters.SearchFilter` to `PostListCreateView`:**

  File: `api/views.py` **Modify:**

  Python

```
# api/views.py
# ... (existing imports) ...
from rest_framework import filters # Import filters

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]

    filter_backends = [DjangoFilterBackend, filters.SearchFilter] #
Add SearchFilter
    filterset_fields = ['owner__username'] # Use this for exact match
filtering
    search_fields = ['title', 'content', 'owner__username'] # Fields
to search across
```

  Explanation:

  - `filters.SearchFilter`: Enables search functionality.
  - `search_fields`: Specifies the fields that the search query should be applied to. By default, the search is case-insensitive and checks for partial matches. You can prefix fields with `^` (starts-with), `=` (exact match), `@` (full-text search), `$` (regex).
- **2. Run the server and test:**
  - `python manage.py runserver`
  - **Test:** Go to `http://127.0.0.1:8000/api/posts/`. You'll see a "Search" input field in the browsable API.
  - **Test with query parameters:**
    - `http://127.0.0.1:8000/api/posts/?search=hello` (Search for "hello" in title, content, or owner username)
    - `http://127.00.0.1:8000/api/posts/?search=Alice` (Search for posts by Alice)

4.6 Filtering based on the date

We can filter posts based on their creation or update dates using `django-filter`'s lookup expressions.

- **1. Enhance `filterset_fields` in `PostListCreateView`:**

  File: `api/views.py` **Modify:**

  Python

  ```
  # api/views.py
  # ... (existing imports) ...

  class PostListCreateView(generics.ListCreateAPIView):
      queryset = Post.objects.all()
      serializer_class = PostSerializer
      permission_classes = [permissions.IsAuthenticatedOrReadOnly]

      filter_backends = [DjangoFilterBackend, filters.SearchFilter]
      filterset_fields = [
          'owner__username',
          'created_at__date', # Filter by exact date (e.g., YYYY-MM-DD)
          'created_at__year', # Filter by year
          'created_at__month', # Filter by month
          'created_at__day', # Filter by day
          'created_at__gt', # Filter posts created after a date/time
          'created_at__lt', # Filter posts created before a date/time
      ]
      search_fields = ['title', 'content', 'owner__username']
  ```

  Explanation:

  - `__date`, `__year`, `__month`, `__day`, `__gt` (greater than), `__lt` (less than) are Django's ORM lookup types. `django-filter` integrates with these.
- **2. Run the server and test:**
  - `python manage.py runserver`
  - **Test with query parameters:**
    - `http://127.0.0.1:8000/api/posts/?created_at__date=2025-07-09` (Assuming you created posts today)
    - `http://127.0.0.1:8000/api/posts/?created_at__year=2025`
    - `http://127.0.0.1:8000/api/posts/?created_at__gt=2025-07-01T00:00:00Z` (Posts after July 1st, 2025)
    - `http://127.0.0.1:8000/api/posts/?created_at__lt=2025-07-10T00:00:00Z` (Posts before July 10th, 2025)

4.7 Ordering Filters

Ordering filters allow clients to specify how the results should be sorted (e.g., by date, title, etc.).

- **1. Add `rest_framework.filters.OrderingFilter` to `PostListCreateView`:**

  File: `api/views.py` **Modify:**

  Python

```
# api/views.py
# ... (existing imports) ...

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]

    filter_backends = [DjangoFilterBackend, filters.SearchFilter,
filters.OrderingFilter] # Add OrderingFilter
    filterset_fields = [
        'owner__username',
        'created_at__date', 'created_at__year', 'created_at__month',
'created_at__day',
        'created_at__gt', 'created_at__lt',
    ]
    search_fields = ['title', 'content', 'owner__username']
    ordering_fields = ['created_at', 'title', 'owner__username'] #
Fields clients can order by
    ordering = ['-created_at'] # Default ordering if client doesn't
specify
```

Explanation:

- `filters.OrderingFilter`: Enables ordering functionality.
- `ordering_fields`: A list of fields that clients can use to sort the results.
- `ordering`: Sets the default sort order for the queryset if no `ordering` parameter is provided in the request. (`-created_at` means descending order by `created_at`).

- **2. Run the server and test:**
  - `python manage.py runserver`
  - **Test:** Go to `http://127.0.0.1:8000/api/posts/`. You'll see a new "Ordering" input in the browsable API.
  - **Test with query parameters:**
    - `http://127.0.0.1:8000/api/posts/?ordering=title` (Order by title ascending)
    - `http://127.0.0.1:8000/api/posts/?ordering=-created_at` (Order by creation date descending, same as default)
    - `http://127.0.0.1:8000/api/posts/?ordering=owner__username` (Order by owner's username ascending)
    - `http://127.0.0.1:8000/api/posts/?ordering=-title,created_at` (Order by title descending, then by creation date ascending for ties)