Good routes versus bad routes

## Introduction

Naming your API properly is the first step in designing a good API. When the API name follows a convention, it provides lots of information about the API and its purpose. To create a meaningful API endpoint, you need to follow some simple guidelines and rules.

In this reading, you will learn about API naming conventions and familiarize yourself with good API endpoints vs. bad API endpoints, or good and bad routes.

## Rule 01: Everything in lowercase, with hyphens and not abridged

The URI of your API should always be in lowercase. Do not use camelCase, PascalCase or Title case when you design your API. Also, separate multiple words using hyphens, not underscores. Do not keep abridged, or shortened, words in your URI; always use the full and meaningful form.

If your API accepts a variable, you should always represent it in camelCase, and wrap it inside a set of curly braces {}.

Let’s examine the following examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /customers | Good | Plural and lowercase |
| /customers/16/phone-number | Good | Lowercase and hyphen used to separate words |
| /customers/16/address/home | Good | Lowercase, no trailing slash, displays the hierarchical relationship with forward slashes. |
| /users/{userId} | Good | Variable in camelCase, and no hyphen in the variable name |
| /Customers | Bad | Title case |
| /generalMembers | Bad | camelCase, no hyphens to separate words |
| /MenuItems  /GeneralMembers | Bad | Pascal case |
| /customers/16/tel-no | Bad | Abbreviation |
| /customers/16/phone\_number | Bad | Underscores |
| /customers/16/phonenumber | Bad | No separation of words |
| /users/{user-id} | Bad | Variable should be in camelCase, and hyphen between words |

## Rule 02: Use a forward slash to indicate a hierarchical relationship

In your API URI, always use the forward slash to indicate a hierarchical relationship. To understand this rule, consider the following scenarios and the relationship from the API endpoints.

A store can have customers who have placed many orders and each of these orders can have delivery addresses, menu items and bills.

| **URI** | **Status** |
| --- | --- |
| /store/customers/{customerId}/orders | Good |
| /store/orders/{orderId} | Good |
| /store/orders/{orderId}/menu-items | Good |

Similarly, a library can have books from many authors. Each of these books has an ISBN number.

| **URI** | **Status** |
| --- | --- |
| /library/authors/books | Good |
| /library/book/{bookId}/isbn | Good |

## Rule 03: Use nouns for resource names, not verbs

One of the most prominent features of REST APIs is that it uses nouns to indicate resources, not verbs. And you should always stick to this rule when designing your API. You should also use plural nouns to indicate a collection.

| **URI** | **Expects** | **Status** | **Why** |
| --- | --- | --- | --- |
| /orders | Collection | Good | Uses a noun, not a verb |
| /users/{userId} | Single user | Good | Uses a noun and proper hierarchical relationship and naming convention |
| /order | Collection | Bad | Uses plural nouns for collections |
| /getOrder | Single resource | Bad | Uses a verb, camelCase |
| /getUser{userId} | Single user | Bad | Uses a verb, camelCase |

## Rule 04: Avoid special characters

You should always avoid special characters in your API endpoints. They can be confusing and technically complex for your users. Consider the following bad examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/12|23|23/address | Bad | Special character | |
| /orders/16/menu^items | Bad | Special character ^ |

If your API can accept multiple user ids, then they should be separated using a comma, as demonstrated below.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/12,23,23/address | Good | Uses a comma for separation |

## Rule 05: Avoid file extensions in URI

You should always avoid file extensions in your API names. For example, if your API can deliver an output in both JSON and XML format, it should never look like this.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /sports/basketball/teams/{teamId}.json | Bad | File extension at the end |
| /sports/basketball/teams/{teamId}.xml | Bad | File extension at the end |

Instead, your client should be able to indicate its expected format in a query string, just like this.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /sports/basketball/teams/{teamId}?format=json | Good | No file extension |
| /sports/basketball/teams/{teamId}?format=xml | Good | No file extension |

Similarly, if your API is serving a static file, for example, CSS or JavaScript files, you should use endpoints like the following to deliver the minified and original source file. You can also use a query string to get the minified or original version. Some API developers use the output format like file extension at the end of the regular API endpoints, which is also bad practice.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /assets/js/jquery/3.12/min | Good | No file extension |
| /assets/js/jquery/3.12/source | Good | No file extension |
| /assets/js/jquery/3.12/?format=min | Good | No file extension |
| /assets/js/jquery/3.12/?format=source | Good | No file extension |
| /menu-items?format=json | Good | Perfectly named endpoint with expected output format in a query string |
| /menu-items.json | Bad | Uses the expected output format as the file extension |

## Rule 06: Use query parameters to filter when necessary

When designing your API, you should always perform data filtering using a query string. This is the same when you expect some extra parameters, like the number of items per page and page number.

Consider this example of a travel site. You want to find which locations a particular user has traveled to. And then you want to know which locations in the USA the user has already seen.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/{userId}/locations | Good | Hierarchical |
| /users/{userId}/locations?country=USA | Good | camelCase, no separation of words |
| /articles?per-page=10&page=2 | Good | Proper use of query string |
| /users/{userId}/locations/USA | Bad | Doesn't use a query string to filter data |
| /articles/page/2/items-per-page/10 | Bad | Redundant and obscure |

## Rule 07: No trailing slash

When sharing your API endpoint with others in your team, or in public, avoid using a trailing slash at the end of your API endpoints. Consider the following examples.

| **URI** | **Status** | **Why** |
| --- | --- | --- |
| /users/{userId} | Good | No trailing slash |
| /articles/{articleId}/author | Good | No trailing slash |
| /users/{userId}/ | Bad | Trailing slash |
| /articles/{articleId}/author/ | Bad | Trailing slash |

## Conclusion

Now you understand how to create REST API endpoints with good names. Remember, a consistent naming strategy for your API is one of the most important design decisions for the whole project!

RESTful APIs are considered to be stateless. What this means is the state is saved

Only with the client.

**REST best practice**s

You should implement caching for your APIs

Kiss(keeping things simple)

filtering, ordering and pagination,

versioning,

cacheing, and rate limiting and monitoring.

**Security and authentication in REST API**

**SSL**(Secure Socket Layer) - SSL encrypts your data and protect your data when it leaves your browser and the web server. When you set up the SSL certificates properly your APIs can be served over HTTPS.

**Signed URLs -** Signed URLs give a client application limited access to a specific resource for a brief period of time. With a signed URL, every time an API is called, a particular piece of text called a signature is included with the URL.

**Authentication versus authorization**

## Introduction

You need to secure your APIs because they provide third-party clients access to your backend data. If you don’t secure your APIs properly, anyone can tamper with the data and access sensitive information. But even if a client is allowed to access the data, you need to control who can do what. This is where authentication and authorization come in. You now know that although they sound similar, they are not the same. In this reading, you will learn about the difference between authentication and authorization and how you can use it to protect your API endpoints.

Authentication

Authentication is the process of verifying the credentials of a user. Logging into websites with a username and password is a typical example of authentication. When the username and password match, the website recognizes the user and sets some cookies in the user’s browser. When the user visits another page on that website, the browser sends those cookies within the HTTP request header. The website recognizes the cookies as well as server-side session data and therefore doesn’t ask for credentials until the user logs out again.

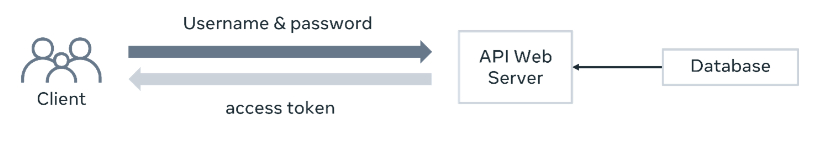
So, how does this work? Token-based authentication usually involves two steps in the API Architecture. First, the client identifies itself with a username and password. Then the API server gives it a bearer token. From there, the client includes the bearer token with every API call that it places. The API server verifies it and then allows the client to perform the action or not. This is where authorization comes in, but more on this later.

If the credentials are not valid, the client will receive a 401 - Unauthorized HTTP status code.

This is like coming to the office on the first day, submitting all your papers and documents, and then receiving your employee card. After that, only your employee card will be sufficient to get inside. Authentication works just like that!

The two steps in the API authentication process can be represented by the following two diagrams.

### **Authentication process: Getting an access token**



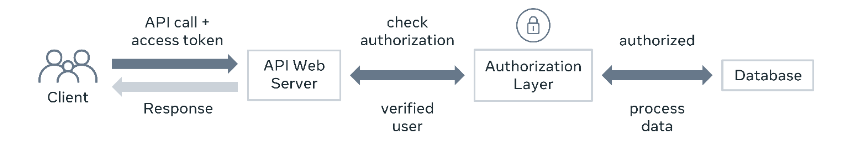
Authenticated API calls

## **Authorization**

However, even with your employee card, you will not be able to access all the rooms or spaces in the office. There are some places that are only accessible to a certain group of people who have been given that privilege. Authorization is exactly like that. Authentication lets you in, authorization lets you act. It checks after authentication if the user has the proper privileges to perform some tasks.

On the server side, this is typically done by assigning the user to a group or a set of groups. Then, after verifying the token, the code checks if the user belongs to the appropriate group to perform that task. If not, the client will receive a 403 - Forbidden HTTP status code.

### **API authorization**

This extra authorization layer in the API architecture ensures that only people with proper privileges can access and modify data. An authorization system in an API project is very important because it prevents data corruption and data breaches.

## **Implementing authorization**

Privileges are the tasks that an API user performs, and they are the building blocks of an authorization layer. First, as an API developer, you identify the required privileges in your project. For example, for a bookshop, there might be the following types of privileges:

* Browse the books
* Add new books
* Edit books
* Delete books
* Place orders

There can be many other privileges like this. And not every user will have every privilege. For instance, regular customers are not allowed to add and edit books, even if they are properly authenticated. Only managers are allowed to perform those operations.

So, after identifying the privileges, you carefully distribute all these privileges into multiple roles. And then, the authorization check is done in the backend code of each API endpoint that requires a user role check. The developer verifies if the user belongs to the appropriate group or roles, and then makes the decision to allow or deny the action.

**Conclusion**

Authentication and authorization are concepts that differ in function and how they are set up in an API architecture. The knowledge you gained in this reading about user groups, roles and privileges lay the groundwork for all the steps that you will learn later on for setting up a proper security layer in your API projects.

**Organizing an API project**

→ When you upgrade an API you should create a new version of your app.

→ You should use a virtual environment for a project so that you can isolate the dependencies.

# **Consequences of a poorly designed API project**

## **Introduction**

Creating a good API project can be challenging. You need to stick to the conventions, write proper error checks in your code, perform security checks, and make sure that your APIs are using processing power and bandwidth optimally. This all takes time and proper planning. But what happens if you don’t properly plan and execute your APIs?

Let’s examine some of the consequences of a poorly designed API project.

## **Data breach**

| **Reasons** | **Consequences** |
| --- | --- |
| Poor security checks in the code, no authentication or authorization checks, improper file permission, and not using SSL | The most significant risk of a poorly designed API project is a data breach. Sensitive data can leak if you don’t have proper security checks in your code or if you didn’t implement proper permissions for the files stored on the server.  Also, if you are not using SSL for your API endpoints, attackers can steal user data before it reaches your API web server.  Such mistakes can cause severe financial damage and trust issues. |

Fix: Add proper security checks in your code and create a solid authorization layer to prevent unauthorized access to your data. Always double-check these sensitive API endpoints before deploying them to production.

## Data corruption

| **Reasons** | **Consequences** |
| --- | --- |
| Poor security, no authentication or authorization checks, absence of data validation and sanitization of input data | Improper security checks and lack of a solid authorization layer can let any user with a valid authentication token access sensitive APIs and modify the data unexpectedly. Also, creating resources without proper validation checks can create malformed data in the database.  Such mistakes can cause severe data corruption and data loss beyond repair. |

Fix: Besides security checks and a solid authorization layer, an API developer must validate and sanitize user data before processing and saving it.

## **Wastage of computing power and memory**

| **Reasons** | **Consequences** |
| --- | --- |
| Unoptimized code, improper business logic, lack of data validation, unoptimized SQL queries or model relationships, lack of database indexes, and no caching. | Poorly written API code can consume unnecessary computing power and memory with unoptimized code, algorithms and business logic. Unoptimized code, lack of proper database indexing and absence of caching can cause a huge load on the database server by running redundant SQL queries, which slows the whole system down.  Such mistakes can end up increasing the cost of your API infrastructure. |

Fix: To avoid this, always spend time optimizing the code and double-checking your database-related code before deploying your APIs to production.

**Wastage of bandwidth**

| **Reasons** | **Consequences** |
| --- | --- |
| Absence of necessary caching header API code, lack of caching policy on the reverse proxy and on the web server, and lack of pagination and filtering. | If your API project doesn’t follow good API development practices like implementing caching, filtering and pagination can cause your APIs to deliver unnecessary data more times than what is required.  Such mistakes can cause bandwidth wasting and end up charging extra bills in your monthly invoice, as well as poor performance from your API endpoints. Besides, the client applications need to spend more resources and time filtering unnecessary data every time. |

Fix: To avoid this, always send proper caching headers with your API responses and implement filtering and pagination features so that the client application can request and receive only what they need.

## Bad user experience

| **Reasons** | **Consequences** |
| --- | --- |
| Not following the proper naming convention, not sending proper HTTP codes, not accepting Accept headers, absence of pagination, sorting, searching and filtering, and lack of proper error checking in code. | It creates a bad user experience. The client application developers must go through extra processing of the API data, extra code to create the final output, and a steeper learning curve to use your API, which was not necessary if the API was designed by following the standard conventions and best practices.  Not accepting the Accept headers means that the API client is not getting the API output in its required format. That will cause bad experiences because clients need extra time and unnecessary code to process the data on their end.  Also, sending wrong HTTP status codes can cause unexpected errors on the client applications and a bad experience for the users who will use those applications. |

Fix: To avoid this, always follow the proper naming convention and implement data filtering, searching, sorting, searching and pagination features for your API endpoints. Always keep proper error checking in the code and write tests so that it doesn’t create unexpected 5XX errors on the server side.

## Breaking client applications

| **Reasons** | **Consequences** |
| --- | --- |
| Not following the proper versioning system | If you don't maintain the proper versioning system for your API project, it can immediately break backward compatibility, and the client application can stop working instantaneously.  The API can cause failure in the current client applications because your new API requires new request data and delivers new responses. So, their old code will not work anymore. They must refactor it and release a new version of their application as soon as possible.  Such disruption can cause a bad reputation and financial damage for both the API and client application developers. |

## Failure to manage the app

| **Reasons** | **Consequences** |
| --- | --- |
| Keeping everything in one big Django app, adding all business logic in the views. | Django apps can become big and become unmanageable over time if you keep adding functionalities in one single app. And then, adding new features or debugging an error will be painful and take extra time and effort.  Also, adding all business logic in the views file can lead to writing redundant code across multiple classes and function-based views.  Failure to manage an app over time leads to bad coding, patching of errors without test coverage and ultimately, poor performance from the APIs. |

Fix: Distribute the features and functionalities to multiple smaller Django apps in a decoupled way. Additionally, put some business logic in the models which can be reused by the other parts of your API project.

## Conclusion

Taking the time to properly design an API project from the start will save you time and effort over the course of a project. The consequences of a poorly designed API affect everyone who uses your API, including the API developers and client application developers.

The knowledge you gained in this reading will hopefully remind you of everything you need to keep in mind to make your future API projects successful.

# XML and JSON response types

## Introduction

When it comes to displaying output, an API developer should always allow the client to request the preferred content type, such as JSON or XML. Clients can do this by supplying an additional header called Accept in the request header. You learned about this in the reading, [HTTP methods, status codes and response types](https://www.coursera.org/learn/apis/supplement/epTtA/http-methods-status-codes-and-response-types). In this way, the client has full flexibility to use the content in the way they want to. In this reading, you’ll learn more about the two data formats that you will encounter the most in requests from clients, JSON and XML.

## Request headers

Client applications need to send Accept request headers with every HTTP request to receive the output in JSON or XML. For example:

| **Response type** | **Request header** |
| --- | --- |
| JSON | Accept: application/json |
| XML | Accept: application/xml  Accept: text/xml |

## Data conversion

In this course, you will build APIs using Django and Django REST framework, also called DRF. DRF has built-in renderers that can convert your data to JSON and represent it in an interactive, browsable API viewer. There are also third-party renderer classes that can convert these data to XML or YAML. You will learn about these features and write actual code later in this course.

#### JSON versus XML

JSON gained popularity because it’s very simple and lightweight. And creating JSON data or parsing it is easier than XML. However, XML has its own advantages too. XML is more readable and supports data attributes that are not possible in JSON. And you can represent complex data in XML and still keep it readable. JSON is very popular among JavaScript developers because they can instantly read and write JSON data just like a regular object.

The table below compares some of the features of both formats.

| **JSON** | **XML** |
| --- | --- |
| JSON or JavaScript Object Notation is a lightweight and dependency-free data format. | XML or Extensible Markup Language is a powerful, tag-based data format. It is similar to HTML. XML data can be fairly complex. |
| The size of JSON data is smaller than XML. So, it takes less bandwidth. | XML data is lengthier than JSON and takes up more bandwidth. |
| JSON data is like keys and values.  {  "author": "Jack London",  "title": "Seawolf"  } | XML is completely tag-based, it does not have key-value pairs like JSON.  <?xml version="1.0" encoding="UTF-8"?>  <root>  <author>Jack London</author>  <title>Seawolf</title>  </root> |
| Representing array elements is simpler in JSON. Here’s an example:  {  "items": [1,2,3,4,5]  } | You can still display array elements in XML but it’s very verbose.  <?xml version="1.0"  encoding="UTF-8"?>  <root>  <items>  <element>1</element>  <element>2</element>  <element>3</element>  <element>4</element>  <element>5</element>  </items>  </root> |
| Generating and parsing JSON data is faster than XML and this conversion process requires less memory and computing power. | Generating and parsing XML is a complex process, and it usually takes more processing power and memory than processing JSON. |
| There is no way to include comments in JSON data. | XML data can include comments. |

## Conclusion

It is clear that JSON and XML both have benefits and limitations. Later in this course, you will learn how to display API output in both formats, but you will mostly use JSON in this course because it’s very simple and lightweight.

# Mock APIs

## Introduction

When developing APIs, it can take a long time before clients can start using them. Fortunately, as an API developer, you can use the mocking technique to provide the same output as a real API endpoint.

A mock API imitates the real API endpoint with fake data so that the client application developers can start development before the actual API is developed. This allows them to develop their apps without waiting for the production APIs to go live.

In mock API outputs, the data structure remains correct, but the values are fake. And you don’t do any kind of business logic process for these mock data. You just return some pre-generated hard-coded data as a response.

## How it works

When you create the mock API endpoints with fake data, both the API developers and the client application developers can evaluate the output and instantly understand what data they will get from an API call or what data they should deliver.

Then, based on these mock APIs endpoints and their output, these developers can start coding the client applications. They don’t need to wait for the API developers to finish implementing these APIs because they already know what type of data they will get.

And finally, when the actual API is ready, the client application developers will change those mock API endpoints with the real endpoints. Client applications are easier to maintain when using mock API tools.

There are two major steps involved in creating Mock APIs. First, you create mock data that you can deliver from the mock API endpoints. And the second step is to create mock API endpoints and serve those data.

## Mock API tools

You can search for Mock API Data Generator in your browser and get plenty of tools that you can use for free. Similarly, searching for Mock API Endpoints will give you access to some handy free services too.

Mockaroo is a popular fake data generator service that you can use for free, which is available from [www.mockaroo.com](http://www.mockaroo.com/)

Mockapi is a popular mock API service that you can use to create mock API endpoints for free, which you can access at <https://mockapi.io/>

## Conclusion

In this reading, you learned how mock APIs can speed up development. Mocking APIs play an essential role in API development because client application developers don't need to wait for the actual API to become live. It minimizes the time of development and reduces the dependencies between API developers and client application developers.

# Better API view with decorators

Benefits of using the API view decorator

→ It turns standard API output into a browsable API interface.

But you must use the Response class to get this view.

→ It includes helpful information in the output about the current API endpoint

In includes helpful information such as which renderer DRF is used to display the API output, what types of data this API accepts, and so on.

→ It allows you to specify which HTTP methods a function should support

You can do this by passing all those HTTP method names as an array to the API view decorator function.

→ It allows you to use other helpful policy decorators like permission\_classes and throttle\_classes

After using this API view decorator, you can apply other policy decorators to a function-based view.

# **Different types of routing in DRF**

## **Introduction**

The Django REST framework provides different ways of URL mapping or routing in an API project. Besides the traditional style of routing, there are other routing techniques that can save you time while developing. In this reading, you are going to learn about both traditional and other techniques.

Note: All the routings should be done in the urls.py file in your Django app.

## **Regular routes**

The code below maps a function from a views.py file to an API endpoint. Don’t forget that you have to import the path function from the django.urls module first.

from django.urls import path

from . import views

urlpatterns = [

path('books’,views.books) ,

]

This URL pattern maps the books function to the /api/books endpoint. You already know how to specify which HTTP methods a function view can serve by supplying them in the api\_decorator function. The following code allows any function view to serve both HTTP GET and POST methods.

@api\_view([‘GET’,’POST’])

## **Routing to a class method**

If you map a specific method from a class, then you need to declare that method as a @staticmethod first. After that, you can map it in the urls.py file. Here’s an example of a class in the views.py file.

class Orders():

@staticmethod

@api\_view()

def listOrders(request):

     return Response({'message':'list of orders'}, 200)

You can also map this listOrders method in the urls.py file as follows.

from django.urls import path

from . import views

urlpatterns = [

path('orders', views.Orders.listOrders)

]

## **Routing class-based views**

You can save a lot of time in DRF by mapping a class that extends the APIview. You don’t need to individually map every method of such classes. In an upcoming video, Function and class-based views, you will learn that such classes have HTTP verb-specific methods inside them. When a class extends APIview or generic views, you can simply map those classes in the urls.py file.

Here’s the code of the class that extends the APIView.

class BookView(APIView):

def get(self, request, pk):

     return Response({"message":"single book with id " + str(pk)}, status.HTTP\_200\_OK)

def put(self, request, pk):

     return Response({"title":request.data.get('title')}, status.HTTP\_200\_OK)

And here is how you map this class in the urls.py file. All you have to do is map the class as a view against an endpoint.

from django.urls import path

from . import views

urlpatterns = [

    path('books/<int:pk>',views.BookView.as\_view())

]

Now you can make HTTP, GET and PUT calls to the /api/books/{bookId} endpoint without issues. If the class has post(), delete() and patch() methods, it will work with HTTP POST, DELETE and PATCH methods too.

## **Routing classes that extend viewsets**

You can have classes that extend the different types of ViewSets in your API project. Just like the classes that extend APIView, these classes also have specific methods used to respond to different types of HTTP requests. Here’s an example of a typical class that extends the viewset.Viewset class.

Class BookView(viewsets.ViewSet):

def list(self, request):

     return Response({"message":"All books"}, status.HTTP\_200\_OK)

def create(self, request):

     return Response({"message":"Creating a book"}, status.HTTP\_201\_CREATED)

def update(self, request, pk=None):

     return Response({"message":"Updating a book"}, status.HTTP\_200\_OK)

def retrieve(self, request, pk=None):

     return Response({"message":"Displaying a book"}, status.HTTP\_200\_OK)

def partial\_update(self, request, pk=None):

        return Response({"message":"Partially updating a book"}, status.HTTP\_200\_OK)

def destroy(self, request, pk=None):

     return Response({"message":"Deleting a book"}, status.HTTP\_200\_OK)

 You can map this class in the urls.py file in your Django app as follows.

urlpatterns = [

path('books', views.BookView.as\_view(

     {

         'get': 'list',

         'post': 'create',

     })

),

    path('books/<int:pk>',views.BookView.as\_view(

     {

         'get': 'retrieve',

         'put': 'update',

         'patch': 'partial\_update',

         'delete': 'destroy',

     })

)

]

Notice how the HTTP verbs are mapped with each method in this class. Also, note that both the list() and retrieve() methods are present. The list() method is used to display all books, and the retrieve() method is used to display a single book.

After this mapping, you can access the http://127.0.0.1:8000/api/books endpoint with GET and POST methods. While you can access the http://127.0.0.1:8000/api/books/1 endpoint with GET, PUT, PATCH and DELETE.

## **Routing with SimpleRouter class in DRF**

If you have a class that extends ViewSets then you can use different types of built-in routers to map those classes in your urls.py file. Doing things this way means you don’t have to map the individual methods as you did in the previous section. Initiate a SimpleRouter object and map the class in the urls.py file in your Django app as follows.

from rest\_framework.routers import SimpleRouter

router = SimpleRouter(trailing\_slash=False)

router.register('books', views.BookView, basename='books')

urlpatterns = router.urls

After mapping, you can access the api/books and api/books/1 endpoints with the same methods as in the previous example.

Did you notice that the argument trailing\_slash=False was passed, instantiating the SimpleRouter object? Without this argument, your API endpoints will have a trailing slash. And, since you don’t want a trailing slash at the end of your API endpoints, you have to pass this argument.

## Routing with DefaultRouter class in DRF

There is another type of router called DefaultRouter which provides an extra benefit over the SimpleRouter. It creates an API root endpoint with a trailing slash that displays all your API endpoints in one place. You can use it this way in the urls.py file.

from rest\_framework.routers import DefaultRouter

router = DefaultRouter(trailing\_slash=False)

router.register('books', views.BookView, basename='books')

urlpatterns = router.urls

Again, after mapping, you can access the api/books and api/books/1 endpoints with the same methods as in the previous examples.

Additionally, you can access the API root view when you visit the http://127.0.0.1:8000/api/ endpoint. This will display all the available endpoints as in the screenshot below.

# **Generic views and ViewSets in DRF**

## Introduction

DRF comes with many generic views and ViewSet to speed up API development. When you use these classes, you don’t need to start from scratch and using them will reduce the code in your API project. In this reading, you will learn about different types of generic views and ViewSet as well as their purposes and benefits.

## ViewSets

ViewSets are simple class-based views, but they come with benefits. There are a few ViewSets classes available in DRF that you can use to quickly scaffold a functioning API CRUD project. You can also provide permission classes and throttle classes to allow authenticated API calls and rate limiting.

To use these classes, you must import the viewsets module from the rest\_framework:

from rest\_framework import viewsets

### ViewSet

There are a few ViewSet classes but the foundation is ViewSet and it extends the APIView. When your class-based views extend a ViewSet you get browsable API views out of the box. Except for that, every ViewSet comes with a method naming convention for easier one-line routing that saves a lot of time.

When a ViewSet is used to deal with a collection of resources, you can write your business logic in list() and create() methods inside this class.

| **Class method** | **Supported HTTP method** | **Purpose** |
| --- | --- | --- |
| list() | GET | Display resource collection |
| create() | POST | Create new resource |

You can use the following methods to write the business logic when a ViewSet deals with a single resource.

| **Class method** | **Supported HTTP method** | **Purpose** |
| --- | --- | --- |
| retrieve() | GET | Display a single resource |
| update() | PUT | Completely replace a single resource with new data |
| partial\_update() | PATCH | Partially update a single resource |
| destroy | DELETE | Delete a single resource |

When you extend a ViewSet, you will have to manually write code to perform the database operations. But there are two more ViewSet classes that can automatically do that for you. This is how you extend a ViewSet class.

class MenuItemViewSet (viewsets.ViewSet)

### **ModelViewSet**

If the class-based view extends a ModelViewSet, it can automatically handle CRUD operations for you. All you must do is give this class a queryset and a serializer, and everything else will be done automatically.  You don’t need to write code for all those database operations anymore. Later in this course, you will see a practical example of using ModelViewSet to write a functioning CRUD API project with only a few lines of code. Here’s an example of how to extend this ViewSet.

class MenuItemView (viewsets.ModelViewSet)

### **ReadOnlyModelViewSet**

As the name suggests, when your class-based views extend a ReadOnlyModelViewSet, it can only display a single resource and resource collection. No write-operation is allowed by such views, so it doesn’t handle POST, PUT, PATCH or DELETE methods.  Here’s an example of extending a ReadOnlyModelViewSet.

class ReadOnlyMenuItemView (viewsets.ReadOnlyModelViewSet)

## **Generic views**

Generic views are another way of quickly writing class-based views to scaffold fully functional CRUD API projects. There are several generic views that offer a particular functionality, like displaying resources or creating a new resource and so on. All you must do is extend these generic views to make your API endpoints work.

To use these generic view classes, you must import the generics module from the rest\_framework.

from rest\_framework import generics

All generic view classes require a queryset and a serializer to work properly.

Here is a list of generic views in DRF and their purposes.

| **Generic view class** | **Supported method** | **Purpose** |
| --- | --- | --- |
| CreateAPIView | POST | Create a new resource |
| ListAPIView | GET | Display resource collection |
| RetrieveAPIView | GET | Display a single resource |
| DestroyAPIView | DELETE | Delete a single resource |
| UpdateAPIView | PUT and PATCH | Replace or partially update a single resource |
| ListCreateAPIView | GET, POST | Display resource collection and create a new resource |
| RetrieveUpdateAPIView | GET, PUT, PATCH | Display a single resource and replace or partially update it |
| RetrieveDestroyAPIView | GET, DELETE | Display a single resource and delete it |
| RetrieveUpdateDestroyAPIView | GET, PUT, PATCH, DELETE | Display, replace or update and delete a single resource |

### Example code

If you want API endpoints to be capable of displaying resource collection and creating a new resource, you have to extend both ListAPIView and CreateAPIView, or just ListCreateAPIView. Both of the following lines of code do the same job.

class MenuItemView (generics.ListAPIView, generics.CreateAPIView)

And

class MenuItemView (generics.ListCreateAPIView)

Just like ModelViewSet, you must give these generic view classes a queryset and a serializer and you don’t need to manually write code to perform these database operations.

## Authentication and selective authentication

If you want all API calls to be authenticated in a class-based view that extends the generic views, you can add the permission\_classes public attribute in the class.

Permission\_classes = [IsAuthenticated]

If you want to selectively enable authentication for some calls, like POST, PUT, PATCH and DELETE then you need to override the get\_permission method in your class-based view like this.

def get\_permissions(self):

        permission\_classes = []

        if self.request.method != 'GET':

            permission\_classes = [IsAuthenticated]

        return [permission() for permission in permission\_classes]

This way, anyone will be able to make GET call, but other HTTP methods like POST, PUT, PATCH and DELETE will require authentication or a valid user token.

## Return items for the authenticated user only

Sometimes in a class-based view that extends a generic view, you may want to return resources created by the authenticated users only. In that case, you need to override the get\_queryset method. The following code in a class-based view returns only those orders created by the authenticated user.

class OrderView(generics.ListCreateAPIView):

    queryset = Order.objects.all()

    serializer\_class = OrderSerializer

    permission\_classes = [IsAuthenticated]

    def get\_queryset(self):

        return Order.objects.all().filter(user=self.request.user)

## **Override default behavior**

Though generic views automate everything, you still have full scope to change the default behavior by overriding any of the default methods. Here is an example that returns a simple static response instead of the resources.

class OrderView(generics.ListCreateAPIView):

    queryset = Order.objects.all()

    serializer\_class = OrderSerializer

    def get(self, request, \*args, \*\*kwargs):

        return Response(‘new response’)

The other methods you can override are post(), put(), patch() and delete().

## Conclusion

In this reading, you explored the ViewSet classes and generic view classes which can help you to scaffold a fully functional CRUD API project in a very short time.

**Function and class-based views**

**→** You can extend the classes and add features at any time

→ There’s less code duplication

→ Write specific methods for each type of HTTP request

→ You don’t need to write as much code like with function-based views.

**Serializers**

Serializers are the most popular feature in DRF this are:

→ You can use them to convert objects into a more readable format, like XML.

→ Serializers can validate data during deserialization, ensuring that your data is clean and consistent.

# **Different types of renderers**

## Introduction

Renderers are the core classes in DRF that display the API output in different formats like JSON and XML. You’ve already learned how to use the Browsable API renderer, JSON renderer, and a third-party renderer called XMLRenderer. In this reading, you are going to learn about a few other useful renderers that you can use in your API projects in DRF.

## TemplateHTMLRenderer

Sometimes, even in an API project, it might be required to display HTML output. For example, if you generate an invoicing API, you need to display the transaction and order details in a nicely formatted way using HTML and CSS. In such cases, DRF’s TemplateHTMLRenderer can help.

### Step 1

Using the TemplateHTMLRenderer, you can pass the data to an HTML file and then display that data using Django’s native templating language called DTL, or Django Templating Language.

To test this TemplateHTMLRenderer the menu items need to be displayed in an HTML file instead of JSON. To use this renderer, you first import it from the rest\_framework.renderers module in the views.py file. You also need to import the renderer\_classes decorator.

from rest\_framework.renderers import TemplateHTMLRenderer

from rest\_framework.decorators import api\_view, renderer\_classes

### Step 2

The second step is to create a new function called menu in the views.py file.

@api\_view()

@renderer\_classes ([TemplateHTMLRenderer])

def menu(request):

    items = MenuItem.objects.select\_related('category').all()

    serialized\_item = MenuItemSerializer(items, many=True)

    return Response({'data':serialized\_item.data}, template\_name='menu-items.html')

Note how the serialized data is passed as context to the HTML template file named menu-items.html. You need to put this HTML file inside the templates directory in your Django app, so the path of this file is: LittleLemon/LittleLemonAPI/templates/menu-item.html

Step 3

The third step is to add the following templating code to this HTML file. This code block accepts the template data and displays them in a HTML table.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Menu Items</title>

</head>

<body>

    <table width="100%" style="text-align: left;">

        <tr>

            <th>Item</th> <!-- item column heading -->

            <th>Price</th> <!-- price column heading -->

            <th>Price After Tax</th> <!-- price after tax column heading -->

            <th>Stock</th> <!-- stock column heading -->

        </tr>

        {% for item in data %}

        <tr>

            <td>{{ item.title }}</td>

            <td>{{ item.price }}</td>

            <td>{{ item.price\_after\_tax }}</td>

            <td>{{ item.stock }}</td>

        </tr>

        {% endfor %}

    </table>

</body>

</html>

### Step 4

The final step is to map this function to an API endpoint in the urls.py file so that it can be browsed as http://127.0.0.1:8000/api/menu.

from django.urls import path

from . import views

urlpatterns = [

    path('menu-items',views.menu\_items),

    path('menu-items/<int:id>',views.single\_item)

    path('menu',views.menu),

]

## **StaticHTMLRenderer**

You can use the StaticHTMLRenderer if any of your API endpoints need to display some HTML content without using any DTL code inside an HTML file.

### Step 1

The first step is to import the StaticHTMLRenderer class and renderer\_classes decorator like before.

from rest\_framework.renderers import TemplateHTMLRenderer

from rest\_framework.decorators import api\_view, renderer\_classes

### Step 2

Then you need to create a new function called welcome in the views.py file.

@api\_view(['GET'])

@renderer\_classes([StaticHTMLRenderer])

def welcome(request):

    data = '<html><body><h1>Welcome To Little Lemon API Project</h1></body></html>'

    return Response(data)

### Step 3

The final step is to map this endpoint to an API endpoint. This time, you want to display this message whenever someone visits the endpoint http://127.0.0.1:8000/api/welcome. To do this, you need to open the urls.py file and add the following line to the urlpatterns list:

path('welcome',views.welcome)

## CSV renderer

CSV, or comma-separated values, is another popular format used by API developers. Unlike JSON or XML, every field in a database record is displayed separated by a comma and every record is on a new line.

### Step 1

DRF doesn’t come with a CSV renderer class by default. So the first step is to install a popular third-party package using pipenv.

pipenv install djangorestframework-csv

### Step 2

Import this renderer in the views.py file.

from rest\_framework\_csv.renderers import CSVRenderer

### Step 3

Add the renderer using the renderer\_classes decorator to convert an API endpoint to display CSV instead of JSON. Add the following line of code in the menu-items function after the @api\_view() decorator:

@renderer\_classes([CSVRenderer])

## YAML renderer

### Step 1

To display the output of your APIs in YAML, another popular data format, you need to install the djangorestframework-yaml using pipenv.

pipenv install djangorestframework-yaml

### Step 2

To test it with the menu-items function, import this YAML renderer in the views.py file.

from rest\_framework\_yaml.renderers import YAMLRenderer

## Step 3

Pass the YAMLRenderer class inside the renderer\_classes decorator, just below the api\_view decorator above the menu-items function.

@renderer\_classes([YAMLRenderer])

## Global settings

Instead of importing the CSV and YAML renderer classes individually in the views.py file and then passing them to the renderer\_classes decorator above each function, you can make them available globally in your API project. In this way, the client can get the desired output with a valid Accept header.

To make these renderers available globally, add the following two lines in the settings.py file in the DEFAULT\_RENDERER\_CLASSES section.

'rest\_framework\_csv.renderers.CSVRenderer',

'rest\_framework\_yaml.renderers.YAMLRenderer',

 This is what the DEFAULT\_RENDERER\_CLASSES section will be like after adding those two lines.

REST\_FRAMEWORK = {

    'DEFAULT\_RENDERER\_CLASSES': [

        'rest\_framework.renderers.JSONRenderer',

        'rest\_framework.renderers.BrowsableAPIRenderer',

        'rest\_framework\_xml.renderers.XMLRenderer',

        'rest\_framework\_csv.renderers.CSVRenderer',

        'rest\_framework\_yaml.renderers.YAMLRenderer',

    ]

}

Now the client can send the following Accept headers to receive the API output in their desired format.

| **Response type** | **Request header** |
| --- | --- |
| CSV | Accept: text/csv |
| YAML | Accept: application/yaml |

## Conclusion

In this reading, you have learned how to use different types of renderers in your DRF-based API project to display API output in different formats. There is a dedicated section on renderers in the DRF documentation, which you can access in the additional resources of this lesson. It showcases other types of renderers you can use with the Django REST Framework.