schemas.py

Schemas

> *A machine-readable [schema] describes what resources are available via the API, what their URLs are, how they are represented and what operations they support.*
>
> — Heroku, *JSON Schema for the Heroku Platform API*

API schemas are a useful tool that allow for a range of use cases, including generating reference documentation, or driving dynamic client libraries that can interact with your API.

Representing schemas internally

REST framework uses Core API in order to model schema information in a format-independent representation. This information can then be rendered into various different schema formats, or used to generate API documentation.

When using Core API, a schema is represented as a `Document` which is the top-level container object for information about the API. Available API interactions are represented using `Link` objects. Each link includes a URL, HTTP method, and may include a list of `Field` instances, which describe any parameters that may be accepted by the API endpoint. The `Link` and `Field` instances may also include descriptions, that allow an API schema to be rendered into user documentation.

Here's an example of an API description that includes a single `search` endpoint:

```
coreapi.Document(
    title='Flight Search API',
```

```
    title='Flight Search API',
    url='https://api.example.org/',
    content={
        'search': coreapi.Link(
            url='/search/',
            action='get',
            fields=[
                coreapi.Field(
                    name='from',
                    required=True,
                    location='query',
                    description='City name or airport code.'
                ),
                coreapi.Field(
                    name='to',
                    required=True,
                    location='query',
                    description='City name or airport code.'
                ),
                coreapi.Field(
                    name='date',
                    required=True,
                    location='query',
                    description='Flight date in "YYYY-MM-DD" format.'
                )
            ],
            description='Return flight availability and prices.'
        )

    }
)
```

Schema output formats

In order to be presented in an HTTP response, the internal representation has to be rendered into the actual bytes that are used in the response.

Core JSON is designed as a canonical format for use with Core API. REST framework includes a renderer class for handling this media type, which is available as `renderers.CoreJSONRenderer`.

Other schema formats such as Open API ("Swagger"), JSON HyperSchema, or API Blueprint can also be supported by implementing a custom renderer class.

Schemas vs Hypermedia

It's worth pointing out here that Core API can also be used to model hypermedia responses, which present an alternative interaction style to API schemas.

With an API schema, the entire available interface is presented up-front as a single endpoint. Responses to individual API endpoints are then typically presented as plain data, without any further interactions contained in each response.

With Hypermedia, the client is instead presented with a document containing both data and available interactions. Each interaction results in a new document, detailing both the current state and the available interactions.

Further information and support on building Hypermedia APIs with REST framework is planned for a future version.

Adding a schema

You'll need to install the `coreapi` package in order to add schema support for REST framework.

```
pip install coreapi
```

REST framework includes functionality for auto-generating a schema, or allows you to specify one explicitly. There are a few different ways to add a schema to your API, depending on exactly what you need.

The get_schema_view shortcut

The simplest way to include a schema in your project is to use the `get_schema_view()` function.

```
schema_view = get_schema_view(title="Server Monitoring API")

urlpatterns = [
    url('^$', schema_view),
    ...
]
```

Once the view has been added, you'll be able to make API requests to retrieve the auto-generated schema definition.

```
$ http http://127.0.0.1:8000/ Accept:application/vnd.coreapi+json
HTTP/1.0 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/vnd.coreapi+json

{
    "_meta": {
        "title": "Server Monitoring API"

    },
    "_type": "document",
    ...
}
```

The arguments to `get_schema_view()` are:

`title`

May be used to provide a descriptive title for the schema definition.

`url`

May be used to pass a canonical URL for the schema.

```
schema_view = get_schema_view(
    title='Server Monitoring API',
    url='https://www.example.org/api/'
)
```

`urlconf`

A string representing the import path to the URL conf that you want to generate an API schema for. This defaults to the value of Django's ROOT_URLCONF setting.

```
schema_view = get_schema_view(
    title='Server Monitoring API',
    url='https://www.example.org/api/',
    urlconf='myproject.urls'
)
```

`renderer_classes`

May be used to pass the set of renderer classes that can be used to render the API root endpoint.

```
from rest_framework.renderers import CoreJSONRenderer
from my_custom_package import APIBlueprintRenderer

schema_view = get_schema_view(
    title='Server Monitoring API',
    url='https://www.example.org/api/',
    renderer_classes=[CoreJSONRenderer, APIBlueprintRenderer]
)
```

Using an explicit schema view

If you need a little more control than the `get_schema_view()` shortcut gives you, then you can use the `SchemaGenerator` class directly to auto-generate the `Document` instance, and to return that from a view.

This option gives you the flexibility of setting up the schema endpoint with whatever behaviour you want. For example, you can apply different permission, throttling, or authentication policies to the schema endpoint.

Here's an example of using `SchemaGenerator` together with a view to return the schema.

views.py:

```
from rest_framework.decorators import api_view, renderer_classes
```

```
from rest_framework import renderers, response, schemas

generator = schemas.SchemaGenerator(title='Bookings API')

@api_view()
@renderer_classes([renderers.CoreJSONRenderer])
def schema_view(request):
    schema = generator.get_schema(request)
    return response.Response(schema)
```

urls.py:

```
urlpatterns = [
    url('/', schema_view),
    ...
]
```

You can also serve different schemas to different users, depending on the permissions they have available. This approach can be used to ensure that unauthenticated requests are presented with a different schema to authenticated requests, or to ensure that different parts of the API are made visible to different users depending on their role.

In order to present a schema with endpoints filtered by user permissions, you need to pass the `request` argument to the `get_schema()` method, like so:

```
@api_view()
@renderer_classes([renderers.CoreJSONRenderer])
def schema_view(request):
    generator = schemas.SchemaGenerator(title='Bookings API')
    return response.Response(generator.get_schema(request=request))
```

Explicit schema definition

An alternative to the auto-generated approach is to specify the API schema explicitly, by declaring a `Document` object in your codebase. Doing so is a little more work, but ensures that you have full control over the schema representation.

```
import coreapi
from rest_framework.decorators import api_view, renderer_classes
from rest_framework import renderers, response

schema = coreapi.Document(
    title='Bookings API',
    content={
        ...
    }
)
```

```
@api_view()
@renderer_classes([renderers.CoreJSONRenderer])
def schema_view(request):
    return response.Response(schema)
```

Static schema file

A final option is to write your API schema as a static file, using one of the available formats, such as Core JSON or Open API.

You could then either:

- Write a schema definition as a static file, and serve the static file directly.
- Write a schema definition that is loaded using `Core API`, and then rendered to one of many available formats, depending on the client request.

Schemas as documentation

One common usage of API schemas is to use them to build documentation pages.

The schema generation in REST framework uses docstrings to automatically populate descriptions in the schema document.

These descriptions will be based on:

- The corresponding method docstring if one exists.
- A named section within the class docstring, which can be either single line or multi-line.
- The class docstring.

Examples

An `APIView`, with an explicit method docstring.

```
class ListUsernames(APIView):
    def get(self, request):
        """
        Return a list of all user names in the system.
        """
        usernames = [user.username for user in User.objects.all()]
        return Response(usernames)
```

A `ViewSet`, with an explict action docstring.

```
class ListUsernames(ViewSet):
    def list(self, request):
        """
        Return a list of all user names in the system.
        """
```

```
        usernames = [user.username for user in User.objects.all()]
        return Response(usernames)
```

A generic view with sections in the class docstring, using single-line style.

```
class UserList(generics.ListCreateAPIView):
    """
    get: List all the users.
    post: Create a new user.
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (IsAdminUser,)
```

A generic viewset with sections in the class docstring, using multi-line style.

```
class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.

    retrieve:
    Return a user instance.

    list:
    Return all users, ordered by most recently joined.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer
```

Alternate schema formats

In order to support an alternate schema format, you need to implement a custom renderer class that handles converting a `Document` instance into a bytestring representation.

If there is a Core API codec package that supports encoding into the format you want to use then implementing the renderer class can be done by using the codec.

Example

For example, the `openapi_codec` package provides support for encoding or decoding to the Open API ("Swagger") format:

```
from rest_framework import renderers
from openapi_codec import OpenAPICodec

class SwaggerRenderer(renderers.BaseRenderer):
```

```
    media_type = 'application/openapi+json'
    format = 'swagger'

    def render(self, data, media_type=None, renderer_context=None):
        codec = OpenAPICodec()
        return codec.dump(data)
```

API Reference

SchemaGenerator

A class that deals with introspecting your API views, which can be used to generate a schema.

Typically you'll instantiate `SchemaGenerator` with a single argument, like so:

```
generator = SchemaGenerator(title='Stock Prices API')
```

Arguments:

- `title` **required** - The name of the API.
- `url` - The root URL of the API schema. This option is not required unless the schema is included under path prefix.
- `patterns` - A list of URLs to inspect when generating the schema. Defaults to the project's URL conf.
- `urlconf` - A URL conf module name to use when generating the schema. Defaults to `settings.ROOT_URLCONF`.

get_schema(self, request)

Returns a `coreapi.Document` instance that represents the API schema.

```
@api_view
@renderer_classes([renderers.CoreJSONRenderer])
def schema_view(request):
    generator = schemas.SchemaGenerator(title='Bookings API')
    return Response(generator.get_schema())
```

The `request` argument is optional, and may be used if you want to apply per-user permissions to the resulting schema generation.

get_links(self, request)

Return a nested dictionary containing all the links that should be included in the API schema.

This is a good point to override if you want to modify the resulting structure of the generated schema, as you can build a new dictionary with a different layout.

get_link(self, path, method, view)

Returns a `coreapi.Link` instance corresponding to the given view.

You can override this if you need to provide custom behaviors for particular views.

get_description(self, path, method, view)

Returns a string to use as the link description. By default this is based on the view docstring as described in the "Schemas as Documentation" section above.

get_encoding(self, path, method, view)

Returns a string to indicate the encoding for any request body, when interacting with the given view. Eg. `'application/json'`. May return a blank string for views that do not expect a request body.

get_path_fields(self, path, method, view):

Return a list of `coreapi.Link()` instances. One for each path parameter in the URL.

get_serializer_fields(self, path, method, view)

Return a list of `coreapi.Link()` instances. One for each field in the serializer class used by the view.

get_pagination_fields(self, path, method, view

Return a list of `coreapi.Link()` instances, as returned by the `get_schema_fields()` method on any pagination class used by the view.

get_filter_fields(self, path, method, view)

Return a list of `coreapi.Link()` instances, as returned by the `get_schema_fields()` method of any filter classes used by the view.

Core API

This documentation gives a brief overview of the components within the `coreapi` package that are used to represent an API schema.

Note that these classes are imported from the `coreapi` package, rather than from the `rest_framework` package.

Document

Represents a container for the API schema.

`title`

A name for the API.

`url`

A canonical URL for the API.

`content`

A dictionary, containing the `Link` objects that the schema contains.

In order to provide more structure to the schema, the `content` dictionary may be nested, typically to a second level. For example:

```
content={
    "bookings": {
        "list": Link(...),
        "create": Link(...),
        ...
    },
    "venues": {
        "list": Link(...),
        ...
    },
    ...
}
```

Link

Represents an individual API endpoint.

`url`

The URL of the endpoint. May be a URI template, such as `/users/{username}/`.

`action`

The HTTP method associated with the endpoint. Note that URLs that support more than one HTTP method, should correspond to a single `Link` for each.

`fields`

A list of `Field` instances, describing the available parameters on the input.

`description`

A short description of the meaning and intended usage of the endpoint.

Field

Represents a single input parameter on a given API endpoint.

`name`

A descriptive name for the input.

`required`

A boolean, indicated if the client is required to included a value, or if the parameter can be omitted.

`location`

Determines how the information is encoded into the request. Should be one of the following strings:

"path"

Included in a templated URI. For example a `url` value of `/products/{product_code}/` could be used together with a `"path"` field, to handle API inputs in a URL path such as `/products/slim-fit-jeans/`.

These fields will normally correspond with named arguments in the project URL conf.

"query"

Included as a URL query parameter. For example `?search=sale`. Typically for `GET` requests.

These fields will normally correspond with pagination and filtering controls on a view.

"form"

Included in the request body, as a single item of a JSON object or HTML form. For example `{"colour": "blue", ...}`. Typically for `POST`, `PUT` and `PATCH` requests. Multiple `"form"` fields may be included on a single link.

These fields will normally correspond with serializer fields on a view.

"body"

Included as the complete request body. Typically for `POST`, `PUT` and `PATCH` requests. No more than one `"body"` field may exist on a link. May not be used together with `"form"` fields.

These fields will normally correspond with views that use `ListSerializer` to validate the request input, or with file upload views.

`encoding`

"application/json"

JSON encoded request content. Corresponds to views using `JSONParser`. Valid only if either one or more `location="form"` fields, or a single `location="body"` field is included on the `Link`.

"multipart/form-data"

Multipart encoded request content. Corresponds to views using `MultiPartParser`. Valid only if one or more `location="form"` fields is included on the `Link`.

"application/x-www-form-urlencoded"

URL encoded request content. Corresponds to views using `FormParser`. Valid only if one or more `location="form"` fields is included on the `Link`.

"application/octet-stream"

Binary upload request content. Corresponds to views using `FileUploadParser`. Valid only if a `location="body"` field is included on the `Link`.

`description`

A short description of the meaning and intended usage of the input field.

Documentation built with **MkDocs**.