Tutorial 2: Requests and Responses

From this point we're going to really start covering the core of REST framework. Let's introduce a couple of essential building blocks.

Request objects

REST framework introduces a `Request` object that extends the regular `HttpRequest`, and provides more flexible request parsing. The core functionality of the `Request` object is the `request.data` attribute, which is similar to `request.POST`, but more useful for working with Web APIs.

```
request.POST  # Only handles form data.  Only works for 'POST' method.
request.data  # Handles arbitrary data.  Works for 'POST', 'PUT' and 'PATCH' methods.
```

Response objects

REST framework also introduces a `Response` object, which is a type of `TemplateResponse` that takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

```
return Response(data)  # Renders to content type as requested by the client.
```

Status codes

Using numeric HTTP status codes in your views doesn't always make for obvious reading, and it's easy to not notice if you get an error code wrong. REST framework provides more explicit identifiers for each status code, such as `HTTP_400_BAD_REQUEST` in the `status` module. It's a good idea to use these throughout rather than using numeric identifiers.

Wrapping API views

REST framework provides two wrappers you can use to write API views.

1. The `@api_view` decorator for working with function based views.
2. The `APIView` class for working with class-based views.

These wrappers provide a few bits of functionality such as making sure you receive `Request` instances in your view, and adding context to `Response` objects so that content negotiation can be performed.

The wrappers also provide behaviour such as returning `405 Method Not Allowed` responses when appropriate, and handling any `ParseError` exceptions that occur when accessing `request.data` with malformed input.

Pulling it all together

Okay, let's go ahead and start using these new components to refactor our views slightly.

```
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Our instance view is an improvement over the previous example. It's a little more concise, and the code now feels very similar to if we were working with the Forms API. We're also using named status codes, which makes the response meanings more obvious.

Here is the view for an individual snippet, in the `views.py` module.

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
```

```
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    elif request.method == 'DELETE':
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

This should all feel very familiar - it is not a lot different from working with regular Django views.

Notice that we're no longer explicitly tying our requests or responses to a given content type. `request.data` can handle incoming `json` requests, but it can also handle other formats. Similarly we're returning response objects with data, but allowing REST framework to render the response into the correct content type for us.

Adding optional format suffixes to our URLs

To take advantage of the fact that our responses are no longer hardwired to a single content type let's add support for format suffixes to our API endpoints. Using format suffixes gives us URLs that explicitly refer to a given format, and means our API will be able to handle URLs such as http://example.com/api/items/4.json.

Start by adding a `format` keyword argument to both of the views, like so.

```
def snippet_list(request, format=None):
```

and

```
def snippet_detail(request, pk, format=None):
```

Now update the `snippets/urls.py` file slightly, to append a set of `format_suffix_patterns` in addition to the existing URLs.

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>', views.snippet_detail),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

We don't necessarily need to add these extra url patterns in, but it gives us a simple, clean way of referring to a specific format.

How's it looking?

Go ahead and test the API from the command line, as we did in tutorial part 1. Everything is working pretty similarly, although we've got some nicer error handling if we send invalid requests.

We can get a list of all of the snippets, as before.

```
http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK
...
[
  {
    "id": 1,
    "title": "",
    "code": "foo = \"bar\"\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  },
  {
    "id": 2,
    "title": "",
    "code": "print(\"hello, world\")\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  }
]
```

We can control the format of the response that we get back, either by using the `Accept` header:

```
http http://127.0.0.1:8000/snippets/ Accept:application/json  # Request JSON
http http://127.0.0.1:8000/snippets/ Accept:text/html         # Request HTML
```

Or by appending a format suffix:

```
http http://127.0.0.1:8000/snippets.json  # JSON suffix
http http://127.0.0.1:8000/snippets.api   # Browsable API suffix
```

Similarly, we can control the format of the request that we send, using the `Content-Type` header.

```
# POST using form data
http --form POST http://127.0.0.1:8000/snippets/ code="print(123)"

{
  "id": 3,
  "title": "",
  "code": "print(123)",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}

# POST using JSON
http --json POST http://127.0.0.1:8000/snippets/ code="print(456)"

{
    "id": 4,
    "title": "",
    "code": "print(456)",
    "linenos": false,
    "language": "python",
    "style": "friendly"
}
```

If you add a `--debug` switch to the `http` requests above, you will be able to see the request type in request headers.

Now go and open the API in a web browser, by visiting http://127.0.0.1:8000/snippets/.

Browsability

Because the API chooses the content type of the response based on the client request, it will, by default, return an HTML-formatted representation of the resource when that resource is requested by a web browser. This allows for the API to return a fully web-browsable HTML representation.

Having a web-browsable API is a huge usability win, and makes developing and using your API much easier. It also dramatically lowers the barrier-to-entry for other developers wanting to inspect and work with your API.

See the browsable api topic for more information about the browsable API feature and how to customize it.

What's next?

In tutorial part 3, we'll start using class-based views, and see how generic views reduce the amount of code we need to write.
