source: test.py

Testing

> Code without tests is broken as designed.
>
> — Jacob Kaplan-Moss

REST framework includes a few helper classes that extend Django's existing test framework, and improve support for making API requests.

APIRequestFactory

Extends Django's existing `RequestFactory` class.

Creating test requests

The `APIRequestFactory` class supports an almost identical API to Django's standard `RequestFactory` class. This means that the standard `.get()`, `.post()`, `.put()`, `.patch()`, `.delete()`, `.head()` and `.options()` methods are all available.

```
from rest_framework.test import APIRequestFactory

# Using the standard RequestFactory API to create a form POST request
factory = APIRequestFactory()
request = factory.post('/notes/', {'title': 'new idea'})
```

Using the `format` argument

Methods which create a request body, such as `post`, `put` and `patch`, include a `format` argument, which make it easy to generate requests using a content type other than multipart form data. For example:

```
# Create a JSON POST request
factory = APIRequestFactory()
request = factory.post('/notes/', {'title': 'new idea'}, format='json')
```

By default the available formats are `'multipart'` and `'json'`. For compatibility with Django's existing `RequestFactory` the default format is `'multipart'`.

To support a wider set of request formats, or change the default format, see the configuration section.

Explicitly encoding the request body

If you need to explicitly encode the request body, you can do so by setting the `content_type` flag. For example:

```
request = factory.post('/notes/', json.dumps({'title': 'new idea'}),
content_type='application/json')
```

PUT and PATCH with form data

One difference worth noting between Django's `RequestFactory` and REST framework's `APIRequestFactory` is that multipart form data will be encoded for methods other than just `.post()`.

For example, using `APIRequestFactory`, you can make a form PUT request like so:

```
factory = APIRequestFactory()
request = factory.put('/notes/547/', {'title': 'remember to email dave'})
```

Using Django's `RequestFactory`, you'd need to explicitly encode the data yourself:

```
from django.test.client import encode_multipart, RequestFactory

factory = RequestFactory()
data = {'title': 'remember to email dave'}
content = encode_multipart('BoUnDaRyStRiNg', data)
content_type = 'multipart/form-data; boundary=BoUnDaRyStRiNg'
request = factory.put('/notes/547/', content, content_type=content_type)
```

Forcing authentication

When testing views directly using a request factory, it's often convenient to be able to directly authenticate the request, rather than having to construct the correct authentication credentials.

To forcibly authenticate a request, use the `force_authenticate()` method.

```
from rest_framework.test import force_authenticate

factory = APIRequestFactory()
user = User.objects.get(username='olivia')
view = AccountDetail.as_view()

# Make an authenticated request to the view...
request = factory.get('/accounts/django-superstars/')
force_authenticate(request, user=user)
response = view(request)
```

The signature for the method is `force_authenticate(request, user=None, token=None)`. When

making the call, either or both of the user and token may be set.

For example, when forcibly authenticating using a token, you might do something like the following:

```
user = User.objects.get(username='olivia')
request = factory.get('/accounts/django-superstars/')
force_authenticate(request, user=user, token=user.token)
```

Note: When using `APIRequestFactory`, the object that is returned is Django's standard `HttpRequest`, and not REST framework's `Request` object, which is only generated once the view is called.

This means that setting attributes directly on the request object may not always have the effect you expect. For example, setting `.token` directly will have no effect, and setting `.user` directly will only work if session authentication is being used.

```
# Request will only authenticate if `SessionAuthentication` is in use.
request = factory.get('/accounts/django-superstars/')
request.user = user
response = view(request)
```

Forcing CSRF validation

By default, requests created with `APIRequestFactory` will not have CSRF validation applied when passed to a REST framework view. If you need to explicitly turn CSRF validation on, you can do so by setting the `enforce_csrf_checks` flag when instantiating the factory.

```
factory = APIRequestFactory(enforce_csrf_checks=True)
```

Note: It's worth noting that Django's standard `RequestFactory` doesn't need to include this option, because when using regular Django the CSRF validation takes place in middleware, which is not run when testing views directly. When using REST framework, CSRF validation takes place inside the view, so the request factory needs to disable view-level CSRF checks.

APIClient

Extends Django's existing `Client` class.

Making requests

The `APIClient` class supports the same request interface as Django's standard `Client` class. This means the that standard `.get()`, `.post()`, `.put()`, `.patch()`, `.delete()`, `.head()` and `.options()` methods are all available. For example:

```
from rest_framework.test import APIClient

client = APIClient()
client.post('/notes/', {'title': 'new idea'}, format='json')
```

To support a wider set of request formats, or change the default format, see the configuration section.

Authenticating

.login(kwargs)**

The `login` method functions exactly as it does with Django's regular `Client` class. This allows you to authenticate requests against any views which include `SessionAuthentication`.

```
# Make all requests in the context of a logged in session.
client = APIClient()
client.login(username='lauren', password='secret')
```

To logout, call the `logout` method as usual.

```
# Log out
client.logout()
```

The `login` method is appropriate for testing APIs that use session authentication, for example web sites which include AJAX interaction with the API.

.credentials(kwargs)**

The `credentials` method can be used to set headers that will then be included on all subsequent requests by the test client.

```
from rest_framework.authtoken.models import Token
from rest_framework.test import APIClient

# Include an appropriate `Authorization:` header on all requests.
token = Token.objects.get(user__username='lauren')
client = APIClient()
client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)
```

```

```

Note that calling `credentials` a second time overwrites any existing credentials. You can unset any existing credentials by calling the method with no arguments.

```
# Stop including any credentials
client.credentials()
```

The `credentials` method is appropriate for testing APIs that require authentication headers, such as basic authentication, OAuth1a and OAuth2 authentication, and simple token authentication schemes.

.force_authenticate(user=None, token=None)

Sometimes you may want to bypass authentication, and simple force all requests by the test client to be automatically treated as authenticated.

This can be a useful shortcut if you're testing the API but don't want to have to construct valid authentication credentials in order to make test requests.

```
user = User.objects.get(username='lauren')
client = APIClient()
client.force_authenticate(user=user)
```

To unauthenticate subsequent requests, call `force_authenticate` setting the user and/or token to `None`.

```
client.force_authenticate(user=None)
```

CSRF validation

By default CSRF validation is not applied when using `APIClient`. If you need to explicitly enable CSRF validation, you can do so by setting the `enforce_csrf_checks` flag when instantiating the client.

```
client = APIClient(enforce_csrf_checks=True)
```

As usual CSRF validation will only apply to any session authenticated views. This means CSRF validation will only occur if the client has been logged in by calling `login()`.

Test cases

REST framework includes the following test case classes, that mirror the existing Django test case classes, but use `APIClient` instead of Django's default `Client`.

- `APISimpleTestCase`
- `APITransactionTestCase`
- `APITestCase`
- `APILiveServerTestCase`

Example

You can use any of REST framework's test case classes as you would for the regular Django test case classes. The `self.client` attribute will be an `APIClient` instance.

```
from django.core.urlresolvers import reverse
from rest_framework import status
from rest_framework.test import APITestCase

class AccountTests(APITestCase):
    def test_create_account(self):
        """
        Ensure we can create a new account object.
        """
        url = reverse('account-list')
        data = {'name': 'DabApps'}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data, data)
```

Testing responses

Checking the response data

When checking the validity of test responses it's often more convenient to inspect the data that the response was created with, rather than inspecting the fully rendered response.

For example, it's easier to inspect `response.data`:

```
response = self.client.get('/users/4/')
self.assertEqual(response.data, {'id': 4, 'username': 'lauren'})
```

Instead of inspecting the result of parsing `response.content`:

```
response = self.client.get('/users/4/')
self.assertEqual(json.loads(response.content), {'id': 4, 'username': 'lauren'})
```

Rendering responses

If you're testing views directly using `APIRequestFactory`, the responses that are returned will not yet be rendered, as rendering of template responses is performed by Django's internal request-response cycle. In order to access `response.content`, you'll first need to render the response.

```
view = UserDetail.as_view()
request = factory.get('/users/4')
response = view(request, pk='4')
response.render()  # Cannot access `response.content` without this.
self.assertEqual(response.content, '{"username": "lauren", "id": 4}')
```

Configuration

Setting the default format

The default format used to make test requests may be set using the `TEST_REQUEST_DEFAULT_FORMAT` setting key. For example, to always use JSON for test requests by default instead of standard multipart form requests, set the following in your `settings.py` file:

```
REST_FRAMEWORK = {
    ...
    'TEST_REQUEST_DEFAULT_FORMAT': 'json'
}
```

Setting the available formats

If you need to test requests using something other than multipart or json requests, you can do so by setting the `TEST_REQUEST_RENDERER_CLASSES` setting.

For example, to add support for using `format='html'` in test requests, you might have something like this in your `settings.py` file.

```
REST_FRAMEWORK = {
    ...
    'TEST_REQUEST_RENDERER_CLASSES': (
        'rest_framework.renderers.MultiPartRenderer',
        'rest_framework.renderers.JSONRenderer',
```

```
        'rest_framework.renderers.TemplateHTMLRenderer'
    )
}
```