source: versioning.py

Versioning

> Versioning an interface is just a "polite" way to kill deployed clients.
>
> — Roy Fielding.

API versioning allows you to alter behavior between different clients. REST framework provides for a number of different versioning schemes.

Versioning is determined by the incoming client request, and may either be based on the request URL, or based on the request headers.

There are a number of valid approaches to approaching versioning. Non-versioned systems can also be appropriate, particularly if you're engineering for very long-term systems with multiple clients outside of your control.

Versioning with REST framework

When API versioning is enabled, the `request.version` attribute will contain a string that corresponds to the version requested in the incoming client request.

By default, versioning is not enabled, and `request.version` will always return `None`.

Varying behavior based on the version

How you vary the API behavior is up to you, but one example you might typically want is to switch to a different serialization style in a newer version. For example:

```
def get_serializer_class(self):
    if self.request.version == 'v1':
        return AccountSerializerVersion1
    return AccountSerializer
```

Reversing URLs for versioned APIs

The `reverse` function included by REST framework ties in with the versioning scheme. You need to make sure to include the current `request` as a keyword argument, like so.

```
from rest_framework.reverse import reverse

reverse('bookings-list', request=request)
```

The above function will apply any URL transformations appropriate to the request version. For example:

- If `NamespacedVersioning` was being used, and the API version was 'v1', then the URL lookup used would be `'v1:bookings-list'`, which might resolve to a URL like `http://example.org/v1/bookings/`.
- If `QueryParameterVersioning` was being used, and the API version was `1.0`, then the returned URL might be something like `http://example.org/bookings/?version=1.0`

Versioned APIs and hyperlinked serializers

When using hyperlinked serialization styles together with a URL based versioning scheme make sure to include the request as context to the serializer.

```
def get(self, request):
    queryset = Booking.objects.all()
    serializer = BookingsSerializer(queryset, many=True, context={'request':
request})
    return Response({'all_bookings': serializer.data})
```

Doing so will allow any returned URLs to include the appropriate versioning.

Configuring the versioning scheme

The versioning scheme is defined by the `DEFAULT_VERSIONING_CLASS` settings key.

```
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.NamespaceVersioning'
}
```

Unless it is explicitly set, the value for `DEFAULT_VERSIONING_CLASS` will be `None`. In this case the `request.version` attribute will always return `None`.

You can also set the versioning scheme on an individual view. Typically you won't need to do this, as it makes more sense to have a single versioning scheme used globally. If you do need to do so, use the `versioning_class` attribute.

```
class ProfileList(APIView):
    versioning_class = versioning.QueryParameterVersioning
```

Other versioning settings

The following settings keys are also used to control versioning:

- `DEFAULT_VERSION`. The value that should be used for `request.version` when no versioning information is present. Defaults to `None`.

- `ALLOWED_VERSIONS`. If set, this value will restrict the set of versions that may be returned by the versioning scheme, and will raise an error if the provided version if not in this set. Note that the value used for the `DEFAULT_VERSION` setting is always considered to be part of the `ALLOWED_VERSIONS` set. Defaults to `None`.
- `VERSION_PARAMETER`. The string that should used for any versioning parameters, such as in the media type or URL query parameters. Defaults to `'version'`.

You can also set your versioning class plus those three values on a per-view or a per-viewset basis by defining your own versioning scheme and using the `default_version`, `allowed_versions` and `version_param` class variables. For example, if you want to use `URLPathVersioning`:

```
from rest_framework.versioning import URLPathVersioning
from rest_framework.views import APIView

class ExampleVersioning(URLPathVersioning):
    default_version = ...
    allowed_versions = ...
    version_param = ...

class ExampleView(APIVIew):
    versioning_class = ExampleVersioning
```

API Reference

AcceptHeaderVersioning

This scheme requires the client to specify the version as part of the media type in the `Accept` header. The version is included as a media type parameter, that supplements the main media type.

Here's an example HTTP request using the accept header versioning style.

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/json; version=1.0
```

In the example request above `request.version` attribute would return the string `'1.0'`.

Versioning based on accept headers is generally considered as best practice, although other styles may be suitable depending on your client requirements.

Using accept headers with vendor media types

Strictly speaking the `json` media type is not specified as including additional parameters. If you are building a well-specified public API you might consider using a vendor media type. To do so, configure your renderers to use a JSON based renderer with a custom media type:

```
class BookingsAPIRenderer(JSONRenderer):
    media_type = 'application/vnd.megacorp.bookings+json'
```

Your client requests would now look like this:

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/vnd.megacorp.bookings+json; version=1.0
```

URLPathVersioning

This scheme requires the client to specify the version as part of the URL path.

```
GET /v1/bookings/ HTTP/1.1
Host: example.com
Accept: application/json
```

Your URL conf must include a pattern that matches the version with a `'version'` keyword argument, so that this information is available to the versioning scheme.

```
urlpatterns = [
    url(
        r'^(?P<version>[v1|v2]+)/bookings/$',
        bookings_list,
        name='bookings-list'
    ),
    url(
        r'^(?P<version>[v1|v2]+)/bookings/(?P<pk>[0-9]+)/$',
        bookings_detail,
        name='bookings-detail'
    )
]
```

NamespaceVersioning

To the client, this scheme is the same as `URLParameterVersioning`. The only difference is how it is configured in your Django application, as it uses URL namespacing, instead of URL keyword arguments.

```
GET /v1/something/ HTTP/1.1
Host: example.com
Accept: application/json
```

With this scheme the `request.version` attribute is determined based on the `namespace` that matches the incoming request path.

In the following example we're giving a set of views two different possible URL prefixes, each under a different namespace:

```
# bookings/urls.py
urlpatterns = [
    url(r'^$', bookings_list, name='bookings-list'),
    url(r'^(?P<pk>[0-9]+)/$', bookings_detail, name='bookings-detail')
]

# urls.py
urlpatterns = [
    url(r'^v1/bookings/', include('bookings.urls', namespace='v1')),
    url(r'^v2/bookings/', include('bookings.urls', namespace='v2'))
]
```

Both `URLParameterVersioning` and `NamespaceVersioning` are reasonable if you just need a simple versioning scheme. The `URLParameterVersioning` approach might be better suitable for small ad-hoc projects, and the `NamespaceVersioning` is probably easier to manage for larger projects.

HostNameVersioning

The hostname versioning scheme requires the client to specify the requested version as part of the hostname in the URL.

For example the following is an HTTP request to the `http://v1.example.com/bookings/` URL:

```
GET /bookings/ HTTP/1.1
Host: v1.example.com
Accept: application/json
```

By default this implementation expects the hostname to match this simple regular expression:

```
^([a-zA-Z0-9]+)\.[a-zA-Z0-9]+\.[a-zA-Z0-9]+$
```

Note that the first group is enclosed in brackets, indicating that this is the matched portion of the hostname.

The `HostNameVersioning` scheme can be awkward to use in debug mode as you will typically be accessing a raw IP address such as `127.0.0.1`. There are various online services which you to access localhost with a custom subdomain which you may find helpful in this case.

Hostname based versioning can be particularly useful if you have requirements to route incoming requests to different servers based on the version, as you can configure different DNS records for different API versions.

QueryParameterVersioning

This scheme is a simple style that includes the version as a query parameter in the URL. For example:

```
GET /something/?version=0.1 HTTP/1.1
Host: example.com
Accept: application/json
```

Custom versioning schemes

To implement a custom versioning scheme, subclass `BaseVersioning` and override the `.determine_version` method.

Example

The following example uses a custom `X-API-Version` header to determine the requested version.

```
class XAPIVersionScheme(versioning.BaseVersioning):
    def determine_version(self, request, *args, **kwargs):
        return request.META.get('HTTP_X_API_VERSION', None)
```

If your versioning scheme is based on the request URL, you will also want to alter how versioned URLs are determined. In order to do so you should override the `.reverse()` method on the class. See the source code for examples.