parsers.py

Parsers

> *Machine interacting web services tend to use more structured formats for sending data than form-encoded, since they're sending more complex data than simple forms*
>
> — *Malcom Tredinnick, Django developers group*

REST framework includes a number of built in Parser classes, that allow you to accept requests with various media types. There is also support for defining your own custom parsers, which gives you the flexibility to design the media types that your API accepts.

How the parser is determined

The set of valid parsers for a view is always defined as a list of classes. When `request.data` is accessed, REST framework will examine the `Content-Type` header on the incoming request, and determine which parser to use to parse the request content.

Note: When developing client applications always remember to make sure you're setting the `Content-Type` header when sending data in an HTTP request.

If you don't set the content type, most clients will default to using `'application/x-www-form-urlencoded'`, which may not be what you wanted.

As an example, if you are sending `json` encoded data using jQuery with the .ajax() method, you should make sure to include the `contentType: 'application/json'` setting.

Setting the parsers

The default set of parsers may be set globally, using the `DEFAULT_PARSER_CLASSES` setting. For example, the following settings would allow only requests with `JSON` content, instead of the default of JSON or form data.

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework.parsers.JSONParser',
    )
}
```

You can also set the parsers used for an individual view, or viewset, using the `APIView` class-based views.

```
from rest_framework.parsers import JSONParser
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    """
    A view that can accept POST requests with JSON content.
    """
    parser_classes = (JSONParser,)

    def post(self, request, format=None):
        return Response({'received data': request.data})
```

Or, if you're using the `@api_view` decorator with function based views.

```
from rest_framework.decorators import api_view
from rest_framework.decorators import parser_classes

@api_view(['POST'])
@parser_classes((JSONParser,))
def example_view(request, format=None):
    """
    A view that can accept POST requests with JSON content.
    """
    return Response({'received data': request.data})
```

API Reference

JSONParser

Parses `JSON` request content.

.media_type: `application/json`

FormParser

Parses HTML form content. `request.data` will be populated with a `QueryDict` of data.

You will typically want to use both `FormParser` and `MultiPartParser` together in order to fully support HTML form data.

.media_type: `application/x-www-form-urlencoded`

MultiPartParser

Parses multipart HTML form content, which supports file uploads. Both `request.data` will be populated with a `QueryDict`.

You will typically want to use both `FormParser` and `MultiPartParser` together in order to fully support HTML form data.

.media_type: `multipart/form-data`

FileUploadParser

Parses raw file upload content. The `request.data` property will be a dictionary with a single key `'file'` containing the uploaded file.

If the view used with `FileUploadParser` is called with a `filename` URL keyword argument, then that argument will be used as the filename.

If it is called without a `filename` URL keyword argument, then the client must set the filename in the `Content-Disposition` HTTP header. For example `Content-Disposition: attachment; filename=upload.jpg`.

.media_type: `*/*`

Notes:

- The `FileUploadParser` is for usage with native clients that can upload the file as a raw data request. For web-based uploads, or for native clients with multipart upload support, you should use the `MultiPartParser` parser instead.
- Since this parser's `media_type` matches any content type, `FileUploadParser` should generally be the only parser set on an API view.
- `FileUploadParser` respects Django's standard `FILE_UPLOAD_HANDLERS` setting, and the `request.upload_handlers` attribute. See the Django documentation for more details.

Basic usage example:

```
# views.py
class FileUploadView(views.APIView):
    parser_classes = (FileUploadParser,)

    def put(self, request, filename, format=None):
        file_obj = request.data['file']
        # ...
        # do some stuff with uploaded file
        # ...
        return Response(status=204)

# urls.py
urlpatterns = [
    # ...
    url(r'^upload/(?P<filename>[^/]+)$', FileUploadView.as_view())
]
```

Custom parsers

To implement a custom parser, you should override `BaseParser`, set the `.media_type` property, and implement the `.parse(self, stream, media_type, parser_context)` method.

The method should return the data that will be used to populate the `request.data` property.

The arguments passed to `.parse()` are:

stream

A stream-like object representing the body of the request.

media_type

Optional. If provided, this is the media type of the incoming request content.

Depending on the request's `Content-Type:` header, this may be more specific than the renderer's `media_type` attribute, and may include media type parameters. For example `"text/plain; charset=utf-8"`.

parser_context

Optional. If supplied, this argument will be a dictionary containing any additional context that may be required to parse the request content.

By default this will include the following keys: `view`, `request`, `args`, `kwargs`.

Example

The following is an example plaintext parser that will populate the `request.data` property with a string representing the body of the request.

```
class PlainTextParser(BaseParser):
    """
    Plain text parser.
    """
    media_type = 'text/plain'

    def parse(self, stream, media_type=None, parser_context=None):
        """
        Simply return a string representing the body of the request.
        """
        return stream.read()
```

Third party packages

The following third party packages are also available.

YAML

REST framework YAML provides YAML parsing and rendering support. It was previously included directly in the REST framework package, and is now instead supported as a third-party package.

Installation & configuration

Install using pip.

```
$ pip install djangorestframework-yaml
```

Modify your REST framework settings.

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework_yaml.parsers.YAMLParser',
    ),
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework_yaml.renderers.YAMLRenderer',
    ),
}
```

XML

REST Framework XML provides a simple informal XML format. It was previously included directly in the REST framework package, and is now instead supported as a third-party package.

Installation & configuration

Install using pip.

```
$ pip install djangorestframework-xml
```

Modify your REST framework settings.

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework_xml.parsers.XMLParser',
    ),
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework_xml.renderers.XMLRenderer',
    ),
}
```

MessagePack

MessagePack is a fast, efficient binary serialization format. Juan Riaza maintains the djangorestframework-msgpack package which provides MessagePack renderer and parser support for REST framework.

CamelCase JSON

djangorestframework-camel-case provides camel case JSON renderers and parsers for REST framework. This allows serializers to use Python-style underscored field names, but be exposed in the API as Javascript-style camel case field names. It is maintained by Vitaly Babiy.

Documentation built with **MkDocs**.