exceptions.py

Exceptions

> *Exceptions… allow error handling to be organized cleanly in a central or high-level place within the program structure.*
>
> — *Doug Hellmann, Python Exception Handling Techniques*

Exception handling in REST framework views

REST framework's views handle various exceptions, and deal with returning appropriate error responses.

The handled exceptions are:

- Subclasses of `APIException` raised inside REST framework.
- Django's `Http404` exception.
- Django's `PermissionDenied` exception.

In each case, REST framework will return a response with an appropriate status code and content-type. The body of the response will include any additional details regarding the nature of the error.

Most error responses will include a key `detail` in the body of the response.

For example, the following request:

```
DELETE http://api.example.com/foo/bar HTTP/1.1
Accept: application/json
```

Might receive an error response indicating that the `DELETE` method is not allowed on that resource:

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 42

{"detail": "Method 'DELETE' not allowed."}
```

Validation errors are handled slightly differently, and will include the field names as the keys in the response. If the validation error was not specific to a particular field then it will use the "non_field_errors" key, or whatever string value has been set for the `NON_FIELD_ERRORS_KEY` setting.

Any example validation error might look like this:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 94

{"amount": ["A valid integer is required."], "description": ["This field may not be blank."
```

Custom exception handling

You can implement custom exception handling by creating a handler function that converts exceptions raised in your API views into response objects. This allows you to control the style of error responses used by your API.

The function must take a pair of arguments, the first is the exception to be handled, and the second is a dictionary containing any extra context such as the view currently being handled. The exception handler function should either return a `Response` object, or return `None` if the exception cannot be handled. If the handler returns `None` then the exception will be re-raised and Django will return a standard HTTP 500 'server error' response.

For example, you might want to ensure that all error responses include the HTTP status code in the body of the response, like so:

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 62

{"status_code": 405, "detail": "Method 'DELETE' not allowed."}
```

In order to alter the style of the response, you could write the following custom exception handler:

```
from rest_framework.views import exception_handler

def custom_exception_handler(exc, context):
    # Call REST framework's default exception handler first,
```

```
    # to get the standard error response.
    response = exception_handler(exc, context)

    # Now add the HTTP status code to the response.
    if response is not None:
        response.data['status_code'] = response.status_code

    return response
```

The context argument is not used by the default handler, but can be useful if the exception handler needs further information such as the view currently being handled, which can be accessed as `context['view']`.

The exception handler must also be configured in your settings, using the `EXCEPTION_HANDLER` setting key. For example:

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'my_project.my_app.utils.custom_exception_handler'
}
```

If not specified, the `'EXCEPTION_HANDLER'` setting defaults to the standard exception handler provided by REST framework:

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler'
}
```

Note that the exception handler will only be called for responses generated by raised exceptions. It will not be used for any responses returned directly by the view, such as the `HTTP_400_BAD_REQUEST` responses that are returned by the generic views when serializer validation fails.

API Reference

APIException

Signature: `APIException()`

The **base class** for all exceptions raised inside an `APIView` class or `@api_view`.

To provide a custom exception, subclass `APIException` and set the `.status_code`, `.default_detail`, and `default_code` attributes on the class.

For example, if your API relies on a third party service that may sometimes be unreachable, you might want to implement an exception for the "503 Service Unavailable" HTTP response code. You could do this like so:

```
from rest_framework.exceptions import APIException

class ServiceUnavailable(APIException):
```

```
class ServiceUnavailable(APIException):
    status_code = 503
    default_detail = 'Service temporarily unavailable, try again later.'
    default_code = 'service_unavailable'
```

Inspecting API exceptions

There are a number of different properties available for inspecting the status of an API exception. You can use these to build custom exception handling for your project.

The available attributes and methods are:

- `.detail` - Return the textual description of the error.
- `.get_codes()` - Return the code identifier of the error.
- `.get_full_details()` - Return both the textual description and the code identifier.

In most cases the error detail will be a simple item:

```
>>> print(exc.detail)
You do not have permission to perform this action.
>>> print(exc.get_codes())
permission_denied
>>> print(exc.get_full_details())
{'message':'You do not have permission to perform this action.','code':'permission_denied'}
```

In the case of validation errors the error detail will be either a list or dictionary of items:

```
>>> print(exc.detail)
{"name":"This field is required.","age":"A valid integer is required."}
>>> print(exc.get_codes())
{"name":"required","age":"invalid"}
>>> print(exc.get_full_details())
{"name":{"message":"This field is required.","code":"required"},"age":{"message":"A valid i
```

ParseError

Signature: `ParseError(detail=None, code=None)`

Raised if the request contains malformed data when accessing `request.data`.

By default this exception results in a response with the HTTP status code "400 Bad Request".

AuthenticationFailed

Signature: `AuthenticationFailed(detail=None, code=None)`

Raised when an incoming request includes incorrect authentication.

By default this exception results in a response with the HTTP status code "401 Unauthenticated", but it may also result in a "403 Forbidden" response, depending on the authentication scheme in use. See the authentication documentation for more details.

NotAuthenticated

Signature: `NotAuthenticated(detail=None, code=None)`

Raised when an unauthenticated request fails the permission checks.

By default this exception results in a response with the HTTP status code "401 Unauthenticated", but it may also result in a "403 Forbidden" response, depending on the authentication scheme in use. See the authentication documentation for more details.

PermissionDenied

Signature: `PermissionDenied(detail=None, code=None)`

Raised when an authenticated request fails the permission checks.

By default this exception results in a response with the HTTP status code "403 Forbidden".

NotFound

Signature: `NotFound(detail=None, code=None)`

Raised when a resource does not exists at the given URL. This exception is equivalent to the standard `Http404` Django exception.

By default this exception results in a response with the HTTP status code "404 Not Found".

MethodNotAllowed

Signature: `MethodNotAllowed(method, detail=None, code=None)`

Raised when an incoming request occurs that does not map to a handler method on the view.

By default this exception results in a response with the HTTP status code "405 Method Not Allowed".

NotAcceptable

Signature: `NotAcceptable(detail=None, code=None)`

Raised when an incoming request occurs with an `Accept` header that cannot be satisfied by any of the available renderers.

By default this exception results in a response with the HTTP status code "406 Not Acceptable".

UnsupportedMediaType

Signature: `UnsupportedMediaType(media_type, detail=None, code=None)`

Raised if there are no parsers that can handle the content type of the request data when accessing `request.data`.

By default this exception results in a response with the HTTP status code "415 Unsupported Media Type".

Throttled

Signature: `Throttled(wait=None, detail=None, code=None)`

Raised when an incoming request fails the throttling checks.

By default this exception results in a response with the HTTP status code "429 Too Many Requests".

ValidationError

Signature: `ValidationError(detail, code=None)`

The `ValidationError` exception is slightly different from the other `APIException` classes:

- The `detail` argument is mandatory, not optional.
- The `detail` argument may be a list or dictionary of error details, and may also be a nested data structure.
- By convention you should import the serializers module and use a fully qualified `ValidationError` style, in order to differentiate it from Django's built-in validation error. For example. `raise serializers.ValidationError('This field must be an integer value.')`

The `ValidationError` class should be used for serializer and field validation, and by validator classes. It is also raised when calling `serializer.is_valid` with the `raise_exception` keyword argument:

```
serializer.is_valid(raise_exception=True)
```

The generic views use the `raise_exception=True` flag, which means that you can override the style of validation error responses globally in your API. To do so, use a custom exception handler, as described above.

By default this exception results in a response with the HTTP status code "400 Bad Request".
