
Qval Documentation

Release 0.4.1

George

Feb 28, 2020

CONTENTS

1	Get started	3
1.1	Basic usage	3
1.2	Framework-specific instructions	6
1.3	Configuration	7
1.4	Qval's API	8
	Python Module Index	17
	Index	19

Qval is a query parameters validation library designed to be used in small projects that require a lot of repetitive parameter validation. In contrast with DRF's [Validators](#) (and other serialization abstractions), **Qval** requires almost no boilerplate. It is built using context managers and mainly targets [Django](#) and [Django Rest Framework \(DRF\)](#), but also supports [Flask](#) and [Falcon](#).

Qval can validate incoming query parameters, convert them to python objects and automatically report errors to the client.

GET STARTED

In order to use Qval in your project, install it with pip:

```
$ pip install qval
```

The usage is as simple as:

```
>>> from qval import validate
>>> with validate({"integer": "10"}, integer=int) as p:
...     print(type(p.integer) is int, p.integer)
True 10
```

For more verbose and clear examples refer to *Basic usage* and *examples* in the [github repository](#).

1.1 Basic usage

Qval provides two ways to validate query parameters:

1. A function called `validate()`:

```
# qval.py
def validate(
    # Request instance. Must implement the request interface or be a
    ↪dictionary.
    request: Union[Request, Dict[str, str]],
    # A Dictionary in the form of (param_name -> `Validator()` object).
    validators: Dict[str, Validator] = None,
    # Provide true if you want to access any other parameters besides the
    ↪configured ones inside the validation context.
    box_all: bool = True,
    # The factories that will be used to convert the parameters to python
    ↪objects.
    **factories: Optional[Callable[[str], object]],
) -> QueryParamValidator:
```

2. A decorator called `@qval()`:

```
# Wrapped view must accept `request` as either first or second argument
def qval(
    # A Dictionary of (parameter -> factory or None)
    factories: Dict[str, Optional[Callable[[str], Any]]],
    # A Dictionary of (parameter -> Validator)
    validators: Dict[str, Validator] = None,
```

(continues on next page)

(continued from previous page)

```

    # Boxing flag. Provide True to access all provided parameters in the
    ↪context manager
    box_all: bool = True,
    # Optional request instance that will be used to obtain the query
    ↪parameters
    request_: Request = None,
):

```

Let's jump to a quick example. Imagine that you have a RESTful calculator with an endpoint called `/api/divide`. You can use `validate()` to automatically convert the parameters to python objects and then validate them:

```

from qval import validate
...
def division_view(request):
    """
    GET /api/divide?
    param a      : int
    param b      : int, nonzero
    param token  : string, length = 12

    Example: GET /api/divide?a=10&b=2&token=abcdefghijkl -> 200, {"answer":
    ↪5}
    """
    # Parameter validation occurs in the context manager.
    # If validation fails or user code throws an error, the context manager
    # will raise InvalidQueryParamException or APIException respectively.
    # In Django Rest Framework, these exceptions will be processed and result
    # in the error codes 400 and 500 on the client side.
    params = (
        # `a` and `b` must be integers.
        # Note: in order to get a nice error message on the client side,
        # you factory should raise either ValueError or TypeError
        validate(request, a=int, b=int)
        # `b` must be anything but zero
        .nonzero("b")
        # The `transform` callable will be applied to the parameter before
    ↪the check.
        # In this case we'll get `token`'s length and check if it is equal
    ↪to 12.
        .eq("token", 12, transform=len)
    )
    # validation starts here
    with params as p:
        return Response({"answer": p.a // p.b})

```

```

// GET /api/divide?a=10&b=2&token=abcdefghijkl
// Browser:
{
  "answer": 5
}

```

Sending `b = 0` to this endpoint will result in the following message on the client side:

```

// GET /api/divide?a=10&b=0&token=abcdefghijkl
{
  "error": "Invalid `b` value: 0."
}

```

(continues on next page)

(continued from previous page)

}

If you have many parameters and custom validators, it's better to use the `@qval()` decorator:

```

from decimal import Decimal
from qval import Validator, QvalValidationError
...

def price_validator(price: int) -> bool:
    """
    A predicate to validate the `price` query parameter.
    Provides a custom error message.
    """
    if price <= 0:
        # If price does not match our requirements, we raise
        ↳QvalValidationError() with a custom message.
        # This exception will be handled in the context manager and will be
        ↳reraised
        # as InvalidQueryParamException() [HTTP 400].
        raise QvalValidationError(f"Price must be greater than zero, got \
        ↳{price}\'.")
    return True

purchase_factories = {"price": Decimal, "item_id": int, "token": None}
purchase_validators = {
    "token": Validator(lambda x: len(x) == 12),
    # Validator(p) can be omitted if there is only one predicate:
    "item_id": lambda x: x >= 0,
    "price": price_validator,
}

# views.py
from qval import qval
from validators import *
...

# Any function or method wrapped with `qval()` must accept `request` as
# either first or second argument, and `params` as last.
@qval(purchase_factories, purchase_validators)
def purchase_view(request, params):
    """
    GET /api/purchase?
    param item_id : int, positive
    param price   : float, greater than zero
    param token   : string, len == 12

    Example: GET /api/purchase?item_id=1&price=5.8&token=abcdefghijkl
    """
    print(f"{params.item_id} costs {params.price}$.")
    ...

```

1.2 Framework-specific instructions

1.2.1 Django Rest Framework

Django Rest Framework works straight out of the box. Simply add `@qval()` to your views or use `validate()` inside.

1.2.2 Django

For Django *without* DRF you may need to add the exception handler to `settings.MIDDLEWARE`. Qval attempts to do it automatically if `DJANGO_SETTINGS_MODULE` is set. Otherwise, you'll see the following message:

```
WARNING:root:Unable to add the APIException middleware to the MIDDLEWARE list. Django
↳ does not
support handling APIException without DRF integration. Define DJANGO_SETTINGS_MODULE
↳ or
add 'qval.framework_integration.HandleAPIExceptionDjango' to the MIDDLEWARE list.
```

Take a look at the plain Django example [here](#).

1.2.3 Flask

If you are using Flask, you will need to setup the exception handlers:

```
from flask import Flask
from qval.framework_integration import setup_flask_error_handlers
...
app = Flask(__name__)
setup_flask_error_handlers(app)
```

Since `request` in Flask is a global object, you may want to curry `@qval()` before usage:

```
from flask import request
from qval import qval_curry

# Firstly, curry `qval()`
qval = qval_curry(request)
...

# Then use it as a decorator.
# Note: you view now must accept `request` as its first argument
@app.route(...)
@qval(...)
def view(request, params):
    ...
```

Check out the full Flask [example](#). You can run the example using the command below:

```
$ PYTHONPATH=. FLASK_APP=examples/flask-example.py flask run
```

1.2.4 Falcon

Similarly to Flask, with Falcon you will need to setup the error handlers:

```
import falcon
from qval.framework_integration import setup_falcon_error_handlers
...
app = falcon.API()
setup_falcon_error_handlers(app)
```

Full Falcon example can be [found](#) in the github repository.

Use the following command to run the app:

```
$ PYTHONPATH=. python examples/falcon-example.py
```

1.3 Configuration

1.3.1 Settings

Qval supports configuration via config files and environmental variables. If `DJANGO_SETTINGS_MODULE` or `SETTINGS_MODULE` is defined, the specified config module will be used. Otherwise, all lookups will be done in `os.environ`.

Supported variables:

- `QVAL_MAKE_REQUEST_WRAPPER = myapp.myfile.my_func`. Customizes the behavior of the `make_request()` function, which is applied to all incoming requests, then the result is passed to `qval.qval.QueryParamValidator`. The provided function must accept `request` and return an object that supports the request interface (see `DummyRequest`).

For example, the following code adds a print to each `make_request()` call:

```
# app/utils.py
def my_wrapper(f):
    @functools.wraps(f)
    def wrapper(request):
        print(f"Received new request: {request}")
        return f(request)
    return wrapper
```

You will also need to execute `export QVAL_MAKE_REQUEST_WRAPPER=app.utils.my_wrapper` in your console or to add it to the config file.

- `QVAL_REQUEST_CLASS = path.to.CustomRequestClass`. `@qval()` will use it to determine which argument is the request. If you have a custom request class that implements `DummyRequest()` interface, provide it with this variable.

1.3.2 Logging

Qval uses the global `log` object when reporting errors. Example error message:

```
An error occurred during the validation or inside of the context: exc `<class
↳'OverflowError'>` ((34, 'Numerical result out of range')).
| Parameters: <QueryDict: {'a': ['2.2324'], 'b': ['30000000']}>
| Body      : b''
| Exception:
Traceback (most recent call last):
  File "<path>/qval/qval.py", line 338, in inner
    return f(*args, params, **kwargs)
  File "<path>/examples/django-example/app/views.py", line 46, in pow_view
    return JsonResponse({"answer": params.a ** params.b})
OverflowError: (34, 'Numerical result out of range')
Internal Server Error: /api/pow
[19/Nov/2018 07:03:15] "GET /api/pow?a=2.2324&b=30000000 HTTP/1.1" 500 102
```

You can disable the logging entirely by calling `log.disable()`.

1.4 Qval's API

Auto-generated documentation of Qval's code.

1.4.1 qval.qval

```
class qval.qval.QueryParamValidator (request: Union[dict, qval.framework_integration.DummyRequest,
rest_framework.request.Request,
django.http.request.HttpRequest,
flask.wrappers.Request, falcon.request.Request], factories: Dict[str, Optional[type]], validators: Dict[str,
Union[Validator, Callable[Any, bool]]] = None, box_all:
bool = True)
```

Bases: `contextlib.AbstractContextManager`

Validates query parameters.

Examples:

```
>>> r = fwk.DummyRequest({"num": "42", "s": "str", "double": "3.14"})
>>> params = QueryParamValidator(r, dict(num=int, s=None, double=float))
>>> with params as p:
...     print(p.num, p.s, p.double, sep=', ')
42, str, 3.14
```

`__enter__()` → `qval.utils.FrozenBox`

Runs validation on the provided request. See `__exit__()` for additional info.

Returns box of validated values.

`__exit__(exc_type, exc_val, exc_tb)`

If occurred exception is not an `InvalidQueryParamException`, the exception will be re-raised as an `APIException`, which will result in the 500 error on the client side.

Parameters

- `exc_type` – exception type

- **exc_val** – exception instance
- **exc_tb** – exception traceback

Returns None

__init__ (*request*: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request], *factories*: Dict[str, Optional[type]], *validators*: Dict[str, Union[Validator, Callable[Any, bool]]] = None, *box_all*: bool = True)

Instantiates the query validator.

Parameters

- **request** – fwk.Request instance
- **factories** – a mapping of {param -> factory}. Providing None as a factory is equivalent to str or lambda x: x, since parameters are stored as strings.
- **validators** – a dictionary of pre-defined validators
- **box_all** – include all params, even if they're not specified in factories

add_predicate (*param*: str, *predicate*: Callable[Any, bool])

Adds a new check for the provided parameter.

Parameters

- **param** – name of the request parameter
- **predicate** – predicate function

Returns None

apply_to_request (*request*: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request]) → qval.qval.QueryParamValidator

Applies the current validation settings to a new request.

Example:

```
>>> from qval.utils import make_request
>>> request = make_request({"a": "77"})
>>> params = QueryParamValidator(request, {"a": int}, {"a": lambda x: x >=
↪70})
>>> with params as p:
...     print(p.a) # Prints 77
77
>>> with params.apply_to_request({"a": "10"}): pass # Error!
Traceback (most recent call last):
...
qval.exceptions.InvalidQueryParamException: ...
```

Parameters **request** – new request instance

Returns new `QueryParamValidator` instance

check (*param*: str, *predicate*: Callable[Any, bool]) → qval.qval.QueryParamValidator

Adds a new check for the provided parameter.

Parameters

- **param** – name of the request parameter

- **predicate** – predicate function

Returns self

eq (*param*: str, *value*: Any, *transform*: Callable[Any, Any] = <function QueryParamValidator.<lambda>>) → qval.qval.QueryParamValidator

Adds the *equality* check for the provided parameter. For example, if *value* = 10, *param* will be tested as `[transform(param) == 10]`.

Parameters

- **param** – name of the request parameter
- **value** – value to compare with
- **transform** – callable that transforms the parameter, default: `lambda x: x`

Returns self

gt (*param*: str, *value*: Any, *transform*: Callable[Any, Any] = <function QueryParamValidator.<lambda>>) → qval.qval.QueryParamValidator

Adds the *greater than* comparison check for provided parameter. For example, if *value* = 10, *param* will be tested as `[transform(param) > 10]`.

Parameters

- **param** – name of the request parameter
- **value** – value to compare with
- **transform** – callable that transforms the parameter, default: `lambda x: x`

Returns self

lt (*param*: str, *value*: Any, *transform*: Callable[Any, Any] = <function QueryParamValidator.<lambda>>) → qval.qval.QueryParamValidator

Adds the *less than* comparison check for the provided parameter. For example, if *value* = 10, *param* will be tested as `[transform(param) < 10]`.

Parameters

- **param** – name of the request parameter
- **value** – value to compare with
- **transform** – callable that transforms the parameter, default: `lambda x: x`

Returns self

nonzero (*param*: str, *transform*: Callable[Any, Any] = <function QueryParamValidator.<lambda>>) → qval.qval.QueryParamValidator

Adds the *nonzero* check for the provided parameter. For example, if *value* = 10, *param* will be tested as `[transform(param) != 0]`.

Parameters

- **param** – name of the request parameter
- **transform** – callable that transforms the parameter, default: `lambda x: x`

Returns self

positive (*param*: str, *transform*: Callable[Any, Any] = <function QueryParamValidator.<lambda>>) → qval.qval.QueryParamValidator

Adds the *greater than zero* comparison check for the provided parameter. Provided *param* will be tested as `[transform(param) > 0]`.

Parameters

- **param** – name of the request parameter
- **transform** – callable that transforms the parameter, default: `lambda x: x`

Returns `self`

property query_params

Returns the dictionary of query parameters.

```
qval.qval.qval (factories: Dict[str, Optional[Callable[str, Any]]], validators: Dict[str, Union[Validator, Callable[Any, bool]]] = None, box_all: bool = True, request_: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request] = None)
```

A decorator that validates query parameters. The wrapped function must accept a request as the first argument (or second if it's a method), and `params` as last.

Parameters

- **factories** – a mapping (parameter, callable [str -> Any])
- **validators** – a mapping (parameter, validator)
- **box_all** – include all parameters in the output dictionary, even if they're not specified in `factories`
- **request** – optional request object that will always be provided to the validator

Returns wrapped function

```
qval.qval.qval_curry (request: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request])
```

Curries `qval()` decorator and provides the given `request` object to the curried function on each call. This is especially handy in Flask, where `request` is global.

Example: .. code-block:: python

```
>>> r = {"num": "42", "s": "str", "double": "3.14"}
>>> qval = qval_curry(r)
>>> @qval({"num": int, "double": float}, None)
... def view(request, extra_param, params):
...     print(params.num, params.double, params.s, extra_param, sep=', ')
>>> view("test")
42, 3.14, str, test
```

Parameters request – request instance

Returns wrapped `qval(..., request_=request)`

```
qval.qval.validate (request: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request], validators: Dict[str, Union[Validator, Callable[Any, bool]]] = None, box_all: bool = True, **factories: Optional[Callable[str, Any]]) -> qval.qval.QueryParamValidator
```

Shortcut for `QueryParamValidator`.

Examples:

```
>>> r = {"num": "42", "s": "str", "double": "3.14"}
>>> with validate(r, num=int, s=None, double=float) as p:
```

(continues on next page)

(continued from previous page)

```
...     print(p.num + p.double, p.s)
45.14 str
```

```
>>> r = {"price": "43.5$", "n_items": "1"}
>>> currency2f = lambda x: float(x[:-1])
>>> params = validate(r, price=currency2f, n_items=int
...                 ).positive("n_items") # n_items must be greater than 0
>>> with params as p:
...     print(p.price, p.n_items)
43.5 1
```

Parameters

- **request** – a request object
- **validators** – a dictionary of validators
- **box_all** – include all parameters in the output dictionary, even if they're not specified in *factories*
- **factories** – a dictionary of callables that create a python object from their parameter

Returns QueryParamValidator instance

1.4.2 qval.validator

exception qval.validator.QvalValidationError

Bases: Exception

The error raised if validation fails. This exception should be used to provide a custom validation error message to the client.

Example:

```
>>> from qval import validate
>>> def f(v: str) -> bool:
...     if not v.isnumeric():
...         raise QvalValidationError(f"Expected a number, got '{v}'")
...     return True
>>> params = validate({"number": "42"}, {"number": f})
>>> with params: pass # OK
>>> with params.apply_to_request({"number": "a string"}): pass
Traceback (most recent call last):
...
qval.exceptions.InvalidQueryParamException: ...
```

class qval.validator.Validator (*predicates: Union[Validator, Callable[Any, bool]])

Bases: object

Validates the given value using the provided predicates.

__call__ (value: Any) → bool

Applies all stored predicates to the given value.

Parameters **value** – value to validate**Returns** True if all checks have passed, False otherwise**Predicate** = typing.Union[_ForwardRef('Validator'), typing.Callable[[typing.Any], bool]]


```
ValidatorType = typing.Union[_ForwardRef('Validator'), typing.Callable[[typing.Any], bool]]
```

```
__init__ (*predicates: Union[Validator, Callable[Any, bool]])
    Instantiates the validator.
```

Parameters `predicates` (`Callable[[Any], bool]`) – predefined predicates

```
add (predicate: Union[Validator, Callable[Any, bool]]) → qval.validator.Validator
    Adds the predicate to the list.
```

Parameters `predicate` – predicate function

Returns `self`

1.4.3 qval.exceptions

```
exception qval.exceptions.InvalidQueryParamException (detail: Union[dict, str], status:
                                                    int)
```

Bases: `rest_framework.exceptions.APIException`

An error thrown when a parameter fails its validation.

```
__init__ (detail: Union[dict, str], status: int)
    Instantiates the exception.
```

Parameters

- **detail** – dict or string with the details
- **status** – status code

1.4.4 qval.utils

```
class qval.utils.ExcLogger
```

Bases: `object`

A class used to report critical errors.

```
>>> from qval.utils import log
>>> log
ExcLogger()
>>> log.is_enabled
True
>>> log.disable()
>>> print(log)
ExcLogger<<Logger qval (WARNING)>>, enabled = false>
```

```
__init__ ()
    Instantiates the logger.
```

Parameters `logger` – a list of loggers

```
disable ()
    Disables logging.
```

Returns `None`

```
enable ()
    Enables logging.
```

Returns `None`

error (*args, **kwargs)
 Shortcut for `log("error", ...)`.

Parameters

- **args** – log args
- **kwargs** – log kwargs

Returns None

property is_enabled
 Returns True if logging is enabled.

log (level: str, *args, **kwargs)
 Logs a new error message on the given level if logging is enabled.

Parameters

- **args** – logger args
- **kwargs** – logger kwargs

Returns None

class `qval.utils.FrozenBox` (dct: Dict[Any, Any])
 Bases: `object`

A frozen dictionary that allows accessing the elements with `.`

Example:

```

>>> box = FrozenBox({"num": 10, "s": "string"})
>>> print(box.num, box.s)
10 string
>>> box["num"] = 404
Traceback (most recent call last):
...
TypeError: 'FrozenBox' object does not support item assignment
>>> box.num = 404
Traceback (most recent call last):
...
TypeError: 'FrozenBox' object does not support attribute assignment
>>> box.num
10
    
```

__init__ (dct: Dict[Any, Any])

Parameters **dct** – the dict to store

`qval.utils.dummify` (request: Union[dict, qval.framework_integration.DummyRequest, rest_framework.request.Request, django.http.request.HttpRequest, flask.wrappers.Request, falcon.request.Request]) → `qval.framework_integration.DummyRequest`

Constructs a `qval.framework_integration.DummyRequest` with the parameters of the given request.

Parameters **request** – any supported request

Returns `DummyRequest (request.<params>)`

```
qval.utils.get_request_params (request: (<class 'dict'>, <class
                                     'qval.framework_integration.DummyRequest'>,
                                     <class 'rest_framework.request.Request'>, <class
                                     'django.http.request.HttpRequest'>, <class
                                     'flask.wrappers.Request'>, <class 'falcon.request.Request'>))
```

Returns a dictionary of the query parameters of the given request.

Parameters `request` – any supported request

Returns dictionary of parameters

```
qval.utils.make_request (request: Union[dict, qval.framework_integration.DummyRequest,
                                       rest_framework.request.Request, django.http.request.HttpRequest,
                                       flask.wrappers.Request, falcon.request.Request]) -> (<class
                                       'dict'>, <class 'qval.framework_integration.DummyRequest'>,
                                       <class 'rest_framework.request.Request'>, <class
                                       'django.http.request.HttpRequest'>, <class 'flask.wrappers.Request'>,
                                       <class 'falcon.request.Request'>)
```

Creates a `qval.framework_integration.DummyRequest` if request is a dictionary, and returns the request itself otherwise.

The behavior of this function can be customized with the `@_make_request()` decorator. Provide the path to your wrapper using `QVAL_MAKE_REQUEST_WRAPPER` in the settings file or set it as an environment variable. The wrapper function must accept `request` as the parameter and return an object that implements the request interface.

For example, the following code adds `print` to each call of the function:

```
# app/utils.py
def my_wrapper(f):
    @functools.wraps(f)
    def wrapper(request):
        print(f"Received new request: {request}")
        return f(request)
    return wrapper
```

Then execute `export QVAL_MAKE_REQUEST_WRAPPER=app.utils.my_wrapper` in your console or simply add it to the config file.

Parameters `request` – dict or request instance

Returns request

1.4.5 qval.framework_integration

```
class qval.framework_integration.DummyRequest (params: Dict[str, str])
```

Bases: object

DummyRequest. Used for compatibility with the supported frameworks.

```
__init__ (params: Dict[str, str])
```

Initialize self. See `help(type(self))` for accurate signature.

property `query_params`

More semantically correct name for request.GET.

```
class qval.framework_integration.HandleAPIExceptionDjango (get_response)
```

Bases: object

`__init__(get_response)`

Initialize self. See `help(type(self))` for accurate signature.

`process_exception` (`_: django.http.request.HttpRequest, exception: Exception`)

`qval.framework_integration.get_module()` → Union[`qval.framework_integration._EnvironSettings`,
`Module`]

Attempts to load the settings module. If none of the supported env variables are defined, returns `_EnvironSettings()` object.

`qval.framework_integration.load_symbol` (`path: object`)

Imports an object using the given path.

Parameters `path` – path to an object, e.g. `my.module.func_1`

Returns loaded symbol

`qval.framework_integration.setup_django_middleware` (`module: Module = None`)

Setups the exception-handling middleware.

Parameters `module` – settings module

Returns None

`qval.framework_integration.setup_falcon_error_handlers` (`api: falcon.API`)

Setups the error handler for `APIException`.

Parameters `api` – `falcon.API`

Returns

`qval.framework_integration.setup_flask_error_handlers` (`app: flask.Flask`)

Setups the error handler for `APIException`.

Parameters `app` – flask app

Returns None

PYTHON MODULE INDEX

q

`qval.exceptions`, 13

`qval.framework_integration`, 15

`qval.qval`, 8

`qval.utils`, 13

`qval.validator`, 12

Symbols

[__call__\(\)](#) (*qval.validator.Validator method*), 12
[__enter__\(\)](#) (*qval.qval.QueryParamValidator method*), 8
[__exit__\(\)](#) (*qval.qval.QueryParamValidator method*), 8
[__init__\(\)](#) (*qval.exceptions.InvalidQueryParamException method*), 13
[__init__\(\)](#) (*qval.framework_integration.DummyRequest method*), 15
[__init__\(\)](#) (*qval.framework_integration.HandleAPIExceptionDjango method*), 15
[__init__\(\)](#) (*qval.qval.QueryParamValidator method*), 9
[__init__\(\)](#) (*qval.utils.ExcLogger method*), 13
[__init__\(\)](#) (*qval.utils.FrozenBox method*), 14
[__init__\(\)](#) (*qval.validator.Validator method*), 13

A

[add\(\)](#) (*qval.validator.Validator method*), 13
[add_predicate\(\)](#) (*qval.qval.QueryParamValidator method*), 9
[apply_to_request\(\)](#) (*qval.qval.QueryParamValidator method*), 9

C

[check\(\)](#) (*qval.qval.QueryParamValidator method*), 9

D

[disable\(\)](#) (*qval.utils.ExcLogger method*), 13
[dummify\(\)](#) (*in module qval.utils*), 14
[DummyRequest](#) (*class in qval.framework_integration*), 15

E

[enable\(\)](#) (*qval.utils.ExcLogger method*), 13
[eq\(\)](#) (*qval.qval.QueryParamValidator method*), 10
[error\(\)](#) (*qval.utils.ExcLogger method*), 13
[ExcLogger](#) (*class in qval.utils*), 13

F

[FrozenBox](#) (*class in qval.utils*), 14

G

[get_module\(\)](#) (*in module qval.framework_integration*), 16
[get_request_params\(\)](#) (*in module qval.utils*), 14
[gt\(\)](#) (*qval.qval.QueryParamValidator method*), 10

H

[HandleAPIExceptionDjango](#) (*class in qval.framework_integration*), 15

I

[InvalidQueryParamException](#), 13
[is_enabled\(\)](#) (*qval.utils.ExcLogger property*), 14

L

[load_symbol\(\)](#) (*in module qval.framework_integration*), 16
[log\(\)](#) (*qval.utils.ExcLogger method*), 14
[lt\(\)](#) (*qval.qval.QueryParamValidator method*), 10

M

[make_request\(\)](#) (*in module qval.utils*), 15

N

[nonzero\(\)](#) (*qval.qval.QueryParamValidator method*), 10

P

[positive\(\)](#) (*qval.qval.QueryParamValidator method*), 10
[Predicate](#) (*qval.validator.Validator attribute*), 12
[process_exception\(\)](#) (*qval.framework_integration.HandleAPIExceptionDjango method*), 16

Q

[query_params\(\)](#) (*qval.framework_integration.DummyRequest property*), 15

`query_params()` (*qval.qval.QueryParamValidator*
property), 11
`QueryParamValidator` (class in *qval.qval*), 8
`qval()` (in module *qval.qval*), 11
`qval.exceptions` (module), 13
`qval.framework_integration` (module), 15
`qval.qval` (module), 8
`qval.utils` (module), 13
`qval.validator` (module), 12
`qval_curry()` (in module *qval.qval*), 11
`QvalValidationError`, 12

S

`setup_django_middleware()` (in module
qval.framework_integration), 16
`setup_falcon_error_handlers()` (in module
qval.framework_integration), 16
`setup_flask_error_handlers()` (in module
qval.framework_integration), 16

V

`validate()` (in module *qval.qval*), 11
`Validator` (class in *qval.validator*), 12
`ValidatorType` (*qval.validator.Validator* attribute),
12