
Flask-JSON Documentation

Release 0.3.5

Sergey Kozlov

Nov 19, 2022

Contents

1	Installation	3
2	Initialization	5
3	Basic usage	7
4	Examples	9
5	Creating JSON responses	11
5.1	Jsonify HTTP errors	13
6	Encoding values	15
6.1	Iterables	15
6.2	Time values	15
6.3	Translation strings	16
6.4	Custom types	16
6.5	Encoding order	17
7	Errors handling	19
8	JSONP support	21
9	Testing	23
10	Configuration	25
11	API	29
11.1	Low-Level API	34
12	Flask-JSON Changelog	37
12.1	0.3.5	37
12.2	0.3.4	37
12.3	0.3.3	37
12.4	0.3.2	37
12.5	0.3.1	37
12.6	0.3.0	38
12.7	0.2.0	38
12.8	0.1	38

12.9 0.0.1	38
Index	39

Flask-JSON is a simple extension that adds better JSON support to Flask application.

It helps to handle JSON-based requests and provides the following features:

- *json_response()* and *@as_json* to generate JSON responses.
- *JsonError* - exception to generate JSON error responses.
- Extended JSON encoding support (see *Creating JSON responses*).
- *JSONP support* with *@as_json_p*.

- *Installation*
- *Initialization*
- *Basic usage*
- *Examples*
- *Creating JSON responses*
 - *Jsonify HTTP errors*
- *Encoding values*
 - *Iterables*
 - *Time values*
 - *Translation strings*
 - *Custom types*
 - *Encoding order*
- *Errors handling*
- *JSONP support*
- *Testing*
- *Configuration*
- *API*
 - *Low-Level API*
- *Flask-JSON Changelog*
 - *0.3.5*
 - *0.3.4*
 - *0.3.3*
 - *0.3.2*
 - *0.3.1*
 - *0.3.0*
 - *0.2.0*
 - *0.1*
 - *0.0.1*

CHAPTER 1

Installation

Install the extension with:

```
$ easy_install Flask-JSON
```

or:

```
$ pip install Flask-JSON
```


CHAPTER 2

Initialization

Before using Flask-JSON features you have to create *FlaskJSON* instance and initialize it with the Flask application instance. As with common Flask extension there are two ways.

First way is to initialize the extension on construction:

```
app = Flask(__name__)
json = FlaskJSON(app)
```

Another way is to postpone initialization and pass Flask application to the `init_app()` method:

```
app = Flask(__name__)
json = FlaskJSON()
...
json.init_app(app)
```

Flask-JSON provides few decorators and you can use them before and after initialization:

```
# Use decorator before initialization.
json = FlaskJSON()

@json.encoder
def custom_encoder(o):
    pass

json.init_app(app)
```

```
# Use decorator after initialization.
json = FlaskJSON(app)

@json.encoder
def custom_encoder(o):
    pass
```


CHAPTER 3

Basic usage

This section provides simple examples of usage with minimum comments just to demonstrate basic features. Next sections describes features more detailed.

First example shows how to use `json_response()`, `@as_json` and `JsonError` to create JSON responses:

```
from datetime import datetime
from flask import Flask, request
from flask_json import FlaskJSON, JsonError, json_response, as_json

app = Flask(__name__)
FlaskJSON(app)

@app.route('/get_time')
def get_time():
    now = datetime.utcnow()
    return json_response(time=now)

@app.route('/increment_value', methods=['POST'])
def increment_value():
    # We use 'force' to skip mimetype checking to have shorter curl command.
    data = request.get_json(force=True)
    try:
        value = int(data['value'])
    except (KeyError, TypeError, ValueError):
        raise JsonError(description='Invalid value.')
    return json_response(value=value + 1)

@app.route('/get_value')
@as_json
def get_value():
    return dict(value=12)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    app.run()
```

Example responses:

```
$ curl http://localhost:5000/get_time
{"status": 200, "time": "2015-04-14T08:44:13.973000"}

$ curl http://localhost:5000/get_value
{"status": 200, "value": 12}

$ curl -X POST --data 'bla' http://localhost:5000/increment_value
{"status": 400, "description": "Not a JSON."}

$ curl -X POST --data '{"value": "txt"}' http://localhost:5000/increment_value
{"status": 400, "description": "Invalid value."}

$ curl -X POST --data '{"value": 41}' http://localhost:5000/increment_value
{"status": 200, "value": 42}
```

In more advanced example we change configuration and set custom error handler:

```
from datetime import datetime
from flask import Flask, request
from flask_json import FlaskJSON, JsonError, json_response

app = Flask(__name__)
json = FlaskJSON(app)

app.config['JSON_ADD_STATUS'] = False
app.config['JSON_DATETIME_FORMAT'] = '%d/%m/%Y %H:%M:%S'

@json.invalid_json_error
def custom_error_handler(e):
    raise JsonError(status=418, hint="RTFM")

# ... the rest is the same as before ...
```

Now responses looks like that:

```
$ curl http://localhost:5000/get_time
{"time": "14/04/2015 09:26:15"}

$ curl -X POST --data 'bla' http://localhost:5000/increment_value
{"hint": "RTFM"}

$ curl -X POST --data '{"value": "txt"}' http://localhost:5000/increment_value
{"description": "Invalid value."}

$ curl -X POST --data '{"value": 41}' http://localhost:5000/increment_value
{"value": 42}
```

CHAPTER 4

Examples

There are few examples available on [GitHub](#).

You also may take a look at [tests](#).

Creating JSON responses

The main purpose of the Flask-JSON extension is to provide a convenient tool for creating JSON responses. This section describes how you can do that.

Most important function to build JSON response is `json_response()`. All other response related features uses it.

With `json_response()` you can:

- Create JSON response by passing keyword arguments:

```
json_response(server_name='norris', available=True)
```

- Create JSON response with arrays or single values (*new in 0.3.2*):

```
json_response(data=[1, 2, 3])
json_response(data=100500)
```

- Specify HTTP status code for response:

```
json_response(status=400, server_name='norris', available=True)
```

- Specify custom HTTP headers for response:

```
json_response(server_name='norris', headers={'X-STATUS': 'ok'})
```

By default `json_response()` adds HTTP status code to the response JSON:

```
{"status": 200, "server_name": "norris"}
```

but you can disable this or change status field name (see [Configuration](#) for more info).

Note what if you use `data_` then HTTP status is not added unless you pass a dictionary.

Another way is to wrap a view with `@as_json` decorator and return json content:

```
FlaskJSON(app)
...

@as_json
def my_view():
    return dict(server_name="norris")

@as_json
def my_view2():
    return [1, 2, 3] # New in 0.3.2
```

The decorator calls `json_response()` internally and provides the same features. You also can return HTTP status and headers:

```
@as_json
def my_view():
    return dict(server_name="norris"), 401, dict(MYHEADER=12)
```

`@as_json` expects the following return values:

```
@as_json
def my_view():
    return json_content
    # or
    return json_content, http_status
    # or
    return json_content, custom_headers
    # or
    return json_content, http_status, custom_headers
    # or
    return json_content, custom_headers, http_status
```

`json_content` may be `None`, in such situation empty JSON response will be generated:

```
@as_json
def my_view():
    do_some_stuff()
```

```
@as_json
def my_view():
    do_some_stuff()
    return None, 400 # same as {}, 400
```

If you return already created JSON response then it will be used as is:

```
@as_json
def my_view():
    do_some_stuff()
    return json_response(some=value)

@as_json
def my_view2():
    do_some_stuff()
    return json_response(_data=[1, 2, 3], headers_={'X-STATUS': 'ok'})
```

And one more way to create JSON response is to raise `JsonError`:


```
def my_view():
    raise JsonError(error_description='Server is down')
```

It will generate HTTP 400 response with JSON payload.

JsonError's constructor has the same signature as *json_response()* so you can force HTTP status and pass custom headers:

```
def my_view():
    raise JsonError(status_=401,
                    headers_=dict(MYHEADER=12, HEADER2='fail'),
                    error_description='Server is down')
```

5.1 Jsonify HTTP errors

JSON_JSONIFY_HTTP_ERRORS option allows to force returning all standard HTTP errors as JSON.

Now response looks this way:

```
$ curl http://localhost:5000
{"description":"The server encountered an internal error and was unable to complete_
↪your request. Either the server is overloaded or there is an error in the_
↪application.", "reason":"Internal Server Error", "status":500}
```

Encoding values

Flask-JSON supports encoding for several types out of the box and also provides few ways to extend it.

6.1 Iterables

Any iterable type will be converted to list value:

```
# set object
json_response(items=set([1, 2, 3]))
# {status=200, items=[1, 2, 3]}

# generator
json_response(items=(x for x in [3, 2, 42]))
# {status=200, items=[3, 2, 42]}

# iterator
json_response(lst=iter([1, 2, 3]))
# {status=200, items=[1, 2, 3]}
```

6.2 Time values

`datetime`, `date` and `time` will be converted to ISO 8601 or custom format depending on configuration:

```
json_response(datetime=datetime(2014, 5, 12, 17, 24, 10),
              date=date(2015, 12, 7),
              time=time(12, 34, 56))
# {
#   "status": 200,
#   "datetime": "2014-05-12T17:24:10",
#   "date": "2015-12-07",
```

(continues on next page)

(continued from previous page)

```
#    "time": "12:34:56"
# }
```

`JSON_*_FORMAT` options allows to change result format.

6.3 Translation strings

`speaklater`'s `_LazyString` is used by `Flask-Babel` and `Flask-BabelEx`.

You can use it in JSON responses too, `_LazyString` will be converted to Unicode string with translation:

```
json_response(item=gettext('bla'))
# {status=200, item='<translation>'}
```

6.4 Custom types

To encode custom types you can implement special methods `__json__()` or `for_json()`:

```
class MyJsonItem(object):
    def __json__(self):
        return '<__json__>'

def view():
    return json_response(item=MyJsonItem())
# {status=200, item='<__json__>'}
```

```
class MyJsonItem(object):
    def for_json(self):
        return '<for_json>'

def view():
    return json_response(item=MyJsonItem())
# {status=200, item='<for_json>'}
```

Note: To enable this approach you have to set `JSON_USE_ENCODE_METHODS` to `True`.

Another way is to use `@encoder` decorator:

```
@json.encoder
def encoder(o):
    if isinstance(o, MyClass):
        return o.to_string()

def view():
    return json_response(value=MyClass())
```

6.5 Encoding order

Flask-JSON calls encoders in the following order:

- User defined `@encoder`.
- **Flask-JSON encoders:**
 - `_LazyString`
 - `iterables`
 - `datetime`
 - `date`
 - `time`
 - `__json__()` method
 - `for_json()` method
- Flask encoders.

CHAPTER 7

Errors handling

Flask-JSON allows you to change default behaviour related to errors handling by using the following decorators:

@invalid_json_error - allows to handle invalid JSON requests:

```
json = FlaskJSON(app)
...

@json.invalid_json_error
def handler(e):
    # e - original exception.
    raise Something
...

def view():
    # This call runs handler() on invalid JSON.
    data = request.get_json()
    ...
```

@error_handler - allows to handle *JsonError* exceptions:

```
json = FlaskJSON(app)
...

@json.error_handler
def error_handler(e):
    # e - JsonError.
    return json_response(401, text='Something wrong.')
```


CHAPTER 8

JSONP support

If you want to generate JSONP responses then you can use `@as_json_p` decorator.

It expects callback name in the URL query and returns response with javascript function call.

Wrapped view must follow the same requirements as for `@as_json`, additionally string value is supported.

Example:

```
from flask import Flask, request
from flask_json import FlaskJSON, as_json_p

app = Flask(__name__)
json = FlaskJSON(app)

app.config['JSON_ADD_STATUS'] = False
app.config['JSON_JSONP_OPTIONAL'] = False

@app.route('/show_message')
def show_message():
    return """
    <!DOCTYPE html>
    <html>
        <body>
            <script type="application/javascript"
                src="%smessage/hello?callback=alert">
            </script>
        </body>
    </html>
    """ % request.host_url

@app.route('/message/<text>')
@as_json_p
def message(text):
```

(continues on next page)

(continued from previous page)

```
    return text

@app.route('/show_quoted')
def show_quoted():
    return """
    <!DOCTYPE html>
    <html>
        <body>
            <script type="application/javascript"
                src="%squote_message?callback=alert">
            </script>
        </body>
    </html>
    """ % request.host_url

@app.route('/quote_message')
@as_json_p
def quote_message():
    return 'Hello, "Sam".'

@app.route('/dict')
@as_json_p
def dict():
    return {'param': 42}

if __name__ == '__main__':
    app.run()
```

Example responses:

```
$ curl http://localhost:5000/message/hello?callback=alert
alert("hello");

$ curl http://localhost:5000/quote_message?callback=alert
alert("Hello, \"Sam\".");

$ curl http://localhost:5000/dict?callback=alert
alert({
  "param": 42
});
```

You may change default `@as_json_p` behaviour with configurations `JSON_JSONP_STRING_QUOTES`, `JSON_JSONP_OPTIONAL` and `JSON_JSONP_QUERY_CALLBACKS`.

Also there is a possibility to set configuration for the specific view via decorator parameters.

Flask-JSON also may help in testing of your JSON API calls. It replaces Flask's `Response` class with custom one if `TESTING` config flag is enabled.

With Flask-JSON response class `JsonTestResponse` you can use `json` attribute. Here is example test project:

```
import unittest
from flask import Flask
from flask_json import json_response, FlaskJSON, JsonTestResponse

def our_app():
    app = Flask(__name__)
    app.test_value = 0
    FlaskJSON(app)

    @app.route('/increment')
    def increment():
        app.test_value += 1
        return json_response(value=app.test_value)

    return app

class OurAppTestCase(unittest.TestCase):
    def setUp(self):
        self.app = our_app()
        self.app.config['TESTING'] = True

        # We have to change response class manually since TESTING flag is
        # set after Flask-JSON initialization.
        self.app.response_class = JsonTestResponse
        self.client = self.app.test_client()

    def test_app(self):
        r = self.client.get('/increment')
```

(continues on next page)

(continued from previous page)

```
# Here is how we can access to JSON.
assert 'value' in r.json
assert r.json['value'] == 1

if __name__ == '__main__':
    unittest.main()
```

CHAPTER 10

Configuration

You can configure Flask-JSON with the following options:

JSON_ADD_STATUS	Put HTTP status field in all JSON responses. Name of the field depends on <i>JSON_STATUS_FIELD_NAME</i> . See <i>json_response()</i> for more info. Default: True.
JSON_STATUS_FIELD_NAME	Name of the field with HTTP status in JSON response. This field is present only if <i>JSON_ADD_STATUS</i> is enabled. See <i>json_response()</i> for more info. Default: status.
JSON_DECODE_ERROR_MESSAGE	Default error response message for the invalid JSON request. If the message is not None and not empty then description field will be added to JSON response. Default: Not a JSON.
JSON_DATETIME_FORMAT	Format for the datetime values in JSON response. Default is ISO 8601: YYYY-MM-DDTHH:MM:SS or YYYY-MM-DDTHH:MM:SS.mmmmmm. Note what it differs from the default Flask behaviour where <i>datetime</i> is represented in RFC1123 format: Wdy, DD Mon YYYY HH:MM:SS GMT.
JSON_DATE_FORMAT	Format for the date values in JSON response. Default is ISO 8601: YYYY-MM-DD.
JSON_TIME_FORMAT	Format for the time values in JSON response. Default is ISO 8601: HH-MM-SS.
JSON_USE_ENCODE_METHODS	Check for <i>__json__()</i> and <i>for_json()</i> object methods while JSON encoding. This allows to support custom objects in JSON response. Default: False.
JSON_JSONP_STRING_QUOTES	If a view returns a string then surround it with extra quotes. Default: True.
JSON_JSONP_OPTIONAL	Make JSONP optional. If no callback is passed then fallback to JSON response as with <i>@as_json</i> . Default: True.
JSON_JSONP_QUERY_CALLBACKS	List of allowed JSONP callback query parameters. Default: ['callback', 'jsonp'].
JSON_JSONIFY_HTTP_ERRORS	Return standard HTTP Errors as JSON instead of HTML by default. Note: this will register custom error handler in Flask. So, this option should be set before the init of Flask-JSON. Default: False.

See `strftime()` and `strptime()` Behavior for more info about time related formats.

This section describes Flask-JSON functions and classes.

class flask_json.**FlaskJSON** (*app=None*)
Flask-JSON extension class.

encoder (*func*)

This decorator allows to set extra JSON encoding step on response building.

JSON encoding order:

- User defined encoding.
- Flask-JSON encoding.
- Flask encoding.

If user defined encoder returns None then default encoders takes place (Flask-JSON and then Flask).

Example

```
json = FlaskJson(app)
...

@json.encoder
def custom_encoder(o):
    if isinstance(o, MyClass):
        return o.to_string()
```

error_handler (*func*)

This decorator allows to set custom handler for the *JsonError* exceptions.

In custom handler you may return `flask.Response` or raise an exception. If user defined handler returns None then default action takes place (generate JSON response from the exception).

Example

```
json = FlaskJson(app)
...

@json.error_handler
def custom_error_handler(e):
    # e is JsonError.
    return json_response(status=401)
```

See also:

`invalid_json_error()`.

`init_app(app)`

Initializes the application with the extension.

Parameters `app` – Flask application object.

`invalid_json_error(func)`

This decorator allows to set custom handler for the invalid JSON requests.

It will be called by the `request.get_json()`.

If the handler returns or raises nothing then Flask-JSON raises `JsonError`.

Example

```
json = FlaskJson(app)
...

@json.invalid_json_error
def invalid_json_error(e):
    raise SomeException
```

By default JSON response will be generated with HTTP 400:

```
{"status": 400, "description": "Not a JSON."}
```

You also may return a value from the handler then it will be used as `request.get_json()` result on errors.

See also:

`JSON_DECODE_ERROR_MESSAGE`

`flask_json.json_response(status_=200, headers_=None, add_status_=None, data_=None, **kwargs)`

Helper function to build JSON response with the given HTTP status and fields(kwargs).

It also puts HTTP status code to the JSON response if `JSON_ADD_STATUS` is True:

```
app.config['JSON_ADD_STATUS'] = True
json_response(test=12)
# {"status": 200, "test": 12}, response HTTP status is 200.

json_response(400, test=12)
# {"status": 400, "test": 12}, response HTTP status is 400.
```

(continues on next page)

(continued from previous page)

```

json_response(status_=401, test=12)
# {"status": 401, "test": 12}, response HTTP status is 401.

app.config['JSON_ADD_STATUS'] = False
json_response(test=12)
# {"test": 12}, response HTTP status is 200.

```

Name of the HTTP status field is configurable and can be changed with *JSON_STATUS_FIELD_NAME*:

```

app.config['JSON_ADD_STATUS'] = True
app.config['JSON_STATUS_FIELD_NAME'] = 'http_status'
json_response(test=12)
# {"http_status": 200, "test": 12}, response HTTP status is 200.

```

If `kwargs` already contains key with the same name as `JSON_STATUS_FIELD_NAME` then its value will be used instead of HTTP status code:

```

app.config['JSON_ADD_STATUS'] = True

json_response(status_=400, status=100500, test=12)
# {"status": 100500, "test": 12}, but response HTTP status is 400.

json_response(status=100500, test=12)
# {"status": 100500, "test": 12}, but response HTTP status is 200.

app.config['JSON_STATUS_FIELD_NAME'] = 'http_status'
json_response(http_status=100500, test=12)
# {"http_status": 100500, "test": 12}, but response HTTP status is 200.

```

You also may add custom headers to the JSON response by passing iterable or dict to *headers_*:

```

# One way.
headers = {'MY-HEADER': value, 'X-EXTRA': 123}
json_response(headers_=headers, test=12)

# Another way (tuple, list, iterable).
headers = (('MY-HEADER', value), ('X-EXTRA', 123))
json_response(headers_=headers, test=12)

```

Parameters

- **status_** – HTTP response status code.
- **headers_** – iterable or dictionary with header values.
- **add_status** – Add status field. If not set then *JSON_ADD_STATUS* is used.
- **data_** – Data to put in result JSON. It can be used instead of `kwargs` or if you want to pass non-dictionary value.
- **kwargs** – keyword arguments to put in result JSON.

Returns Response with the JSON content.

Return type `flask.Response`

Note: Only `data_` or `kwargs` is allowed.

If `data_` is not a `dict` then `add_status_` and `JSON_ADD_STATUS` are ignored and no status is stored in the result JSON.

If `dict` is passed via `data_` then behaviour is like you pass data in the keyword arguments.

Changed in version 0.3.2: Added `data_` and non-dictionary values support.

`flask_json.as_json(f)`

This decorator converts view's return value to JSON response.

The decorator expects the following return values:

- Flask `Response` instance (see note below);
- a `dict` with JSON content;
- a tuple of (`dict`, `status`) or (`dict`, `headers`) or (`dict`, `status`, `headers`) or (`dict`, `headers`, `status`).

Instead of `dict` you may pass `None` and it will be treated as empty JSON (same as `dict()` or `{}`).

In all other cases it raises an error.

The decorator provides the same features as `json_response()`.

Usage:

```
@as_json
def view_simple():
    return dict(param=value, param2=value2)

@as_json
def view_simple2():
    return [1, 2, 3]

@as_json
def view_comp():
    return dict(param=value, param2=value2), 400
```

Note: If wrapped view returns Flask `Response` then it will be used as is without passing to `json_response()`. But the response must be a JSON response (mimetype must contain `application/json`), otherwise `AssertionError` will be raised.

Returns Response with the JSON content.

Return type `flask.Response`

Raises `ValueError` – if return value is not supported.

See also:

`json_response()`

`flask_json.as_json_p(f=None, callbacks=None, optional=None, add_quotes=None)`

This decorator acts like `@as_json` but also handles JSONP requests; expects string or any `@as_json` supported return value.

It may be used in two forms:

- Without parameters - then global configuration will be applied:

```
@as_json_p
def view():
    ...
```

- With parameters - then they will have priority over global ones for the given view:

```
@as_json_p(...)
def view():
    ...
```

Strings may be surrounded with quotes depending on configuration (`add_quotes` or `JSON_JSONP_STRING_QUOTES`):

```
...
@as_json_p
def view():
    return 'str'

app.config['JSON_JSONP_STRING_QUOTES'] = False
# view() -> callback(str);

app.config['JSON_JSONP_STRING_QUOTES'] = True
# view() -> callback("str");
```

Note: If view returns custom headers or HTTP status then they will be discarded. Also HTTP status field will not be passed to the callback.

Parameters

- **callbacks** – List of acceptable callback query parameters.
- **optional** – Make JSONP optional. If no callback is passed then fallback to JSON response.
- **add_quotes** – If view returns a string then surround it with extra quotes.

Returns JSONP response with javascript function call.

Return type `flask.Response`

Raises

- `ValueError` – if return value is not supported.
- `BadRequest` – if callback is missing in URL query (if `optional=False`).

See also:

`json_response()`, `@as_json`.

`JSON_JSONP_STRING_QUOTES`, `JSON_JSONP_OPTIONAL`, `JSON_JSONP_QUERY_CALLBACKS`.

class `flask_json.JsonError` (`status_=400`, `headers_=None`, `**kwargs`)

Exception which will be converted to JSON response.

Usage:

```
raise JsonError(description='text')
raise JsonError(status_=401, one='text', two=12)
```

`__init__` (*status_=400, headers_=None, **kwargs*)

Construct error object.

Parameters are the same as for `json_response()`.

Parameters

- **status_** – HTTP response status code.
- **headers_** – iterable or dictionary with header values.
- **kwargs** – keyword arguments to put in result JSON.

See also:

`json_response()`, `@error_handler`.

```
class flask_json.JsonTestResponse (response: Union[Iterable[bytes], bytes, Iterable[str], str,
None] = None, status: Union[int, str, http.HTTPStatus,
None] = None, headers: Union[Mapping[str, Union[str,
int, Iterable[Union[str, int]]]], Iterable[Tuple[str, Union[str,
int]]], None] = None, mimetype: Optional[str] = None, content_type: Optional[str] = None, direct_passthrough: bool
= False)
```

JSON Response class for testing.

It provides convenient access to JSON content without explicit response data decoding.

Flask-JSON replaces Flask's response class with this one on initialization if testing mode enabled.

Usage:

```
app = Flask()
app.config['TESTING'] = True
FlaskJSON(app)
...
client = app.test_client()
r = client.get('/view') # suppose it returns json_response(param='12')
assert r.json['param'] == 12
```

If you enable testing after Flask-JSON initialization the you have to set `JsonTestResponse` by yourself:

```
app = Flask()
FlaskJSON(app)
app.config['TESTING'] = True
app.response_class = JsonTestResponse
```

json

Response JSON content.

11.1 Low-Level API

```
class flask_json.JsonRequest (environ: WSGIEnvironment, populate_request: bool = True, shallow: bool = False)
```

This class changes `flask.Request` behaviour on JSON parse errors.

`flask.Request.get_json()` will raise `JsonError` by default on invalid JSON content.

See also:

`JSON_DECODE_ERROR_MESSAGE`, `@invalid_json_error`

on_json_loading_failed(*e*)

Called if `get_json()` fails and isn't silenced.

If this method returns a value, it is used as the return value for `get_json()`. The default implementation raises `BadRequest`.

Parameters *e* – If parsing failed, this is the exception. It will be `None` if the content type wasn't application/json.

class `flask_json.JSONEncoderEx`(***kwargs*)

Extends default Flask JSON encoder with more types:

- iterable;
- `datetime`;
- `date`;
- `time`;
- `speakeater` lazy strings;
- objects with `__json__()` or `for_json()` methods.

Time related values will be converted to ISO 8601 format by default.

See also:

`JSON_DATETIME_FORMAT`, `JSON_DATE_FORMAT`, `JSON_TIME_FORMAT`,
`JSON_USE_ENCODE_METHODS`.

default(*o*)

Convert *o* to a JSON serializable type. See `json.JSONEncoder.default()`. Python does not support overriding how basic types like `str` or `list` are serialized, they are handled before this method.

12.1 0.3.5

- Improve empty data processing ([#9](https://github.com/skozlovf/flask-json/issues/9)).

12.2 0.3.4

- Fix deprecation warning.
- Update tests.

12.3 0.3.3

- Allow jsonify standard HTTP errors.

12.4 0.3.2

- Add non-dictionary values support.
- Fix tests.

12.5 0.3.1

- Switch to pytest.

12.6 0.3.0

- JSONP support.
- Allow to control HTTP status filed in `json_response()`.

12.7 0.2.0

- Support `None` and response instances in `@as_json`.
- Fix `@as_json` documentation.

12.8 0.1

This release is not fully backwards compatible with the 0.0.1 version.

- New `@as_json` decorator.
- New `JsonTestResponse` class.
- Allow to change HTTP status field name.
- Allow to set custom JSON response headers.
- Better JSON error class API.
- Better encoding: more types out of the box, better time values format handling, fixed encoding order.
- Better project documentation.
- Better tests.

Incompatible changes:

- `JsonErrorResponse` renamed to `JsonError`.
- Changed `json_response()` signature.

12.9 0.0.1

First public alpha.

Symbols

`__init__()` (*flask_json.JsonError method*), 33

A

`as_json()` (*in module flask_json*), 32

`as_json_p()` (*in module flask_json*), 32

D

`default()` (*flask_json.JSONEncoderEx method*), 35

E

`encoder()` (*flask_json.FlaskJSON method*), 29

`error_handler()` (*flask_json.FlaskJSON method*),
29

F

`FlaskJSON` (*class in flask_json*), 29

I

`init_app()` (*flask_json.FlaskJSON method*), 30

`invalid_json_error()` (*flask_json.FlaskJSON method*), 30

J

`json` (*flask_json.JsonTestResponse attribute*), 34

`json_response()` (*in module flask_json*), 30

`JSONEncoderEx` (*class in flask_json*), 35

`JsonError` (*class in flask_json*), 33

`JsonRequest` (*class in flask_json*), 34

`JsonTestResponse` (*class in flask_json*), 34

O

`on_json_loading_failed()`
(*flask_json.JsonRequest method*), 34