Flask-Cors Documentation

Release 3.0.9

Cory Dolphin

Sep 06, 2020

Contents

1 Installation **3**

2 Usage **5**
2.1 Simple Usage 5

3 Documentation **7**

4 Troubleshooting **9**

5 Tests **11**

6 Contributing **13**

7 Credits **15**
7.1 API Docs 15

Index **25**

A Flask extension for handling Cross Origin Resource Sharing (CORS), making cross-origin AJAX possible.

This package has a simple philosophy, when you want to enable CORS, you wish to enable it for all use cases on a domain. This means no mucking around with different allowed headers, methods, etc. By default, submission of cookies across domains is disabled due to the security implications, please see the documentation for how to enable credential'ed requests, and please make sure you add some sort of CSRF protection before doing so!

CHAPTER 1

Installation

Install the extension with using pip, or easy_install.

```
$ pip install -U flask-cors
```

CHAPTER 2

Usage

This package exposes a Flask extension which by default enables CORS support on all routes, for all origins and methods. It allows parameterization of all CORS headers on a per-resource level. The package also contains a decorator, for those who prefer this approach.

2.1 Simple Usage

In the simplest case, initialize the Flask-Cors extension with default arguments in order to allow CORS for all domains on all routes. See the full list of options in the documentation.

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

@app.route("/")
def helloWorld():
  return "Hello, cross-origin-world!"
```

2.1.1 Resource specific CORS

Alternatively, you can specify CORS options on a resource and origin level of granularity by passing a dictionary as the *resources* option, mapping paths to a set of options. See the full list of options in the documentation.

```
app = Flask(__name__)
cors = CORS(app, resources={r"/api/*": {"origins": "*"}})

@app.route("/api/v1/users")
def list_users():
  return "user example"
```

2.1.2 Route specific CORS via decorator

This extension also exposes a simple decorator to decorate flask routes with. Simply add `@cross_origin()` below a call to Flask's `@app.route(..)` to allow CORS on a given route. See the full list of options in the decorator documentation.

```
@app.route("/")
@cross_origin()
def helloWorld():
  return "Hello, cross-origin-world!"
```

CHAPTER 3

Documentation

For a full list of options, please see the full documentation

CHAPTER 4

Troubleshooting

If things aren't working as you expect, enable logging to help understand what is going on under the hood, and why.

```
logging.getLogger('flask_cors').level = logging.DEBUG
```

CHAPTER 5

Tests

A simple set of tests is included in `test/`. To run, install nose, and simply invoke `nosetests` or `python setup.py test` to exercise the tests.

CHAPTER 6

Contributing

Questions, comments or improvements? Please create an issue on Github, tweet at @corydolphin or send me an email. I do my best to include every contribution proposed in any way that I can.

CHAPTER 7

Credits

This Flask extension is based upon the Decorator for the HTTP Access Control written by Armin Ronacher.

7.1 API Docs

This package exposes a Flask extension which by default enables CORS support on all routes, for all origins and methods. It allows parameterization of all CORS headers on a per-resource level. The package also contains a decorator, for those who prefer this approach.

7.1.1 Extension

This is the suggested approach to enabling CORS. The default configuration will work well for most use cases.

class flask_cors.**CORS**(*app=None*, ***kwargs*)

Initializes Cross Origin Resource sharing for the application. The arguments are identical to *cross_origin()*, with the addition of a *resources* parameter. The resources parameter defines a series of regular expressions for resource paths to match and optionally, the associated options to be applied to the particular resource. These options are identical to the arguments to *cross_origin()*.

The settings for CORS are determined in the following order

1. Resource level settings (e.g when passed as a dictionary)
2. Keyword argument settings
3. App level configuration settings (e.g. CORS_*)
4. Default settings

Note: as it is possible for multiple regular expressions to match a resource path, the regular expressions are first sorted by length, from longest to shortest, in order to attempt to match the most specific regular expression. This allows the definition of a number of specific resource options, with a wildcard fallback for all other resources.

Parameters

- **resources** (*dict, iterable or string*) – The series of regular expression and (optionally) associated CORS options to be applied to the given resource path.

  If the argument is a dictionary, it's keys must be regular expressions, and the values must be a dictionary of kwargs, identical to the kwargs of this function.

  If the argument is a list, it is expected to be a list of regular expressions, for which the app-wide configured options are applied.

  If the argument is a string, it is expected to be a regular expression for which the app-wide configured options are applied.

  Default : Match all and apply app-level configuration

- **origins** (*list, string or regex*) – The origin, or list of origins to allow requests from. The origin(s) may be regular expressions, case-sensitive strings, or else an asterisk

  Default : '*'

- **methods** (*list or string*) – The method or list of methods which the allowed origins are allowed to access for non-simple requests.

  Default : [GET, HEAD, POST, OPTIONS, PUT, PATCH, DELETE]

- **expose_headers** (*list or string*) – The header or list which are safe to expose to the API of a CORS API specification.

  Default : None

- **allow_headers** (*list, string or regex*) – The header or list of header field names which can be used when this resource is accessed by allowed origins. The header(s) may be regular expressions, case-sensitive strings, or else an asterisk.

  Default : '*', allow all headers

- **supports_credentials** (*bool*) – Allows users to make authenticated requests. If true, injects the *Access-Control-Allow-Credentials* header in responses. This allows cookies and credentials to be submitted across domains.

  note This option cannot be used in conjuction with a '*' origin

  Default : False

- **max_age** (*timedelta, integer, string or None*) – The maximum time for which this CORS request maybe cached. This value is set as the *Access-Control-Max-Age* header.

  Default : None

- **send_wildcard** (*bool*) – If True, and the origins parameter is *, a wildcard *Access-Control-Allow-Origin* header is sent, rather than the request's *Origin* header.

  Default : False

- **vary_header** (*bool*) – If True, the header Vary: Origin will be returned as per the W3 implementation guidelines.

  Setting this header when the *Access-Control-Allow-Origin* is dynamically generated (e.g. when there is more than one allowed origin, and an Origin than '*' is returned) informs CDNs and other caches that the CORS headers are dynamic, and cannot be cached.

  If False, the Vary header will never be injected or altered.

  Default : True

7.1.2 Decorator

If the *CORS* extension does not satisfy your needs, you may find the decorator useful. It shares options with the extension, and should be simple to use.

`flask_cors.`**`cross_origin`**`(*args, **kwargs)`

This function is the decorator which is used to wrap a Flask route with. In the simplest case, simply use the default parameters to allow all origins in what is the most permissive configuration. If this method modifies state or performs authentication which may be brute-forced, you should add some degree of protection, such as Cross Site Forgery Request protection.

Parameters

- **`origins`** (*`list, string or regex`*) – The origin, or list of origins to allow requests from. The origin(s) may be regular expressions, case-sensitive strings, or else an asterisk

  Default : '*'

- **`methods`** (*`list or string`*) – The method or list of methods which the allowed origins are allowed to access for non-simple requests.

  Default : [GET, HEAD, POST, OPTIONS, PUT, PATCH, DELETE]

- **`expose_headers`** (*`list or string`*) – The header or list which are safe to expose to the API of a CORS API specification.

  Default : None

- **`allow_headers`** (*`list, string or regex`*) – The header or list of header field names which can be used when this resource is accessed by allowed origins. The header(s) may be regular expressions, case-sensitive strings, or else an asterisk.

  Default : '*', allow all headers

- **`supports_credentials`** (*`bool`*) – Allows users to make authenticated requests. If true, injects the *Access-Control-Allow-Credentials* header in responses. This allows cookies and credentials to be submitted across domains.

  note This option cannot be used in conjuction with a '*' origin

  Default : False

- **`max_age`** (*`timedelta, integer, string or None`*) – The maximum time for which this CORS request maybe cached. This value is set as the *Access-Control-Max-Age* header.

  Default : None

- **`send_wildcard`** (*`bool`*) – If True, and the origins parameter is *, a wildcard *Access-Control-Allow-Origin* header is sent, rather than the request's *Origin* header.

  Default : False

- **`vary_header`** (*`bool`*) – If True, the header Vary: Origin will be returned as per the W3 implementation guidelines.

  Setting this header when the *Access-Control-Allow-Origin* is dynamically generated (e.g. when there is more than one allowed origin, and an Origin than '*' is returned) informs CDNs and other caches that the CORS headers are dynamic, and cannot be cached.

  If False, the Vary header will never be injected or altered.

  Default : True

- **automatic_options** (*bool*) – Only applies to the *cross_origin* decorator. If True, Flask-CORS will override Flask's default OPTIONS handling to return CORS headers for OPTIONS requests.

  Default : True

7.1.3 Using *CORS* with cookies

By default, Flask-CORS does not allow cookies to be submitted across sites, since it has potential security implications. If you wish to enable cross-site cookies, you may wish to add some sort of CSRF protection to keep you and your users safe.

To allow cookies or authenticated requests to be made cross origins, simply set the *supports_credentials* option to *True*. E.G.

```
from flask import Flask, session
from flask_cors import CORS

app = Flask(__name__)
CORS(app, supports_credentials=True)

@app.route("/")
def helloWorld():
  return "Hello, %s" % session['username']
```

The above code enables Flask backend to accept cookies to be submitted from cross origin sites. But if you are sending Xhr requests (ajax calls) to a cross-origin server, by default chrome or any modern browser won't send cookies and session with the request. You should use `withCredentials = True` while sending Xhr request to enable that. You should keep in mind about the necessary security concerns. Related MDN doc: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials

7.1.4 Using *CORS* with Blueprints

Flask-CORS supports blueprints out of the box. Simply pass a *blueprint* instance to the CORS extension, and everything will just work.

```
api_v1 = Blueprint('API_v1', __name__)

CORS(api_v1) # enable CORS on the API_v1 blue print

@api_v1.route("/api/v1/users/")
def list_users():
    '''
        Since the path matches the regular expression r'/api/*', this resource
        automatically has CORS headers set. The expected result is as follows:

        $ curl --include -X GET http://127.0.0.1:5000/api/v1/users/ \
            --header Origin:www.examplesite.com
        HTTP/1.0 200 OK
        Access-Control-Allow-Headers: Content-Type
        Access-Control-Allow-Origin: *
        Content-Length: 21
        Content-Type: application/json
        Date: Sat, 09 Aug 2014 00:26:41 GMT
        Server: Werkzeug/0.9.4 Python/2.7.8
```

(continues on next page)

(continued from previous page)

```
        {
            "success": true
        }

    '''
    return jsonify(user="joe")


@api_v1.route("/api/v1/users/create", methods=['POST'])
def create_user():
    '''
        Since the path matches the regular expression r'/api/*', this resource
        automatically has CORS headers set.

        Browsers will first make a preflight request to verify that the resource
        allows cross-origin POSTs with a JSON Content-Type, which can be simulated
        as:
        $ curl --include -X OPTIONS http://127.0.0.1:5000/api/v1/users/create \
            --header Access-Control-Request-Method:POST \
            --header Access-Control-Request-Headers:Content-Type \
            --header Origin:www.examplesite.com
        >> HTTP/1.0 200 OK
        Content-Type: text/html; charset=utf-8
        Allow: POST, OPTIONS
        Access-Control-Allow-Origin: *
        Access-Control-Allow-Headers: Content-Type
        Access-Control-Allow-Methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT
        Content-Length: 0
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:22 GMT


        $ curl --include -X POST http://127.0.0.1:5000/api/v1/users/create \
            --header Content-Type:application/json \
            --header Origin:www.examplesite.com


        >> HTTP/1.0 200 OK
        Content-Type: application/json
        Content-Length: 21
        Access-Control-Allow-Origin: *
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:04 GMT

        {
          "success": true
        }

    '''
    return jsonify(success=True)

public_routes = Blueprint('public', __name__)

@public_routes.route("/")
def helloWorld():
    '''
```

(continues on next page)

(continued from previous page)

```
        Since the path '/' does not match the regular expression r'/api/*',
        this route does not have CORS headers set.
    '''
    return '''<h1>Hello CORS!</h1> Read about my spec at the
<a href="http://www.w3.org/TR/cors/">W3</a> Or, checkout my documentation
on <a href="https://github.com/corydolphin/flask-cors">Github</a>'''


logging.basicConfig(level=logging.INFO)
app = Flask('FlaskCorsBlueprintBasedExample')
app.register_blueprint(api_v1)
app.register_blueprint(public_routes)


if __name__ == "__main__":
    app.run(debug=True)
```

7.1.5 Examples

Using the *CORS* extension

```
# One of the simplest configurations. Exposes all resources matching /api/* to
# CORS and allows the Content-Type header, which is necessary to POST JSON
# cross origin.
CORS(app, resources=r'/api/*')


@app.route("/")
def helloWorld():
    """
        Since the path '/' does not match the regular expression r'/api/*',
        this route does not have CORS headers set.
    """
    return '''
<html>
    <h1>Hello CORS!</h1>
    <h3> End to end editable example with jquery! </h3>
    <a class="jsbin-embed" href="http://jsbin.com/zazitas/embed?js,console">JS Bin on␣
↪jsbin.com</a>
    <script src="//static.jsbin.com/js/embed.min.js?3.35.12"></script>

</html>
'''

@app.route("/api/v1/users/")
def list_users():
    """
        Since the path matches the regular expression r'/api/*', this resource
        automatically has CORS headers set. The expected result is as follows:

        $ curl --include -X GET http://127.0.0.1:5000/api/v1/users/ \
            --header Origin:www.examplesite.com
        HTTP/1.0 200 OK
        Access-Control-Allow-Headers: Content-Type
```

(continues on next page)

(continued from previous page)

```
        Access-Control-Allow-Origin: *
        Content-Length: 21
        Content-Type: application/json
        Date: Sat, 09 Aug 2014 00:26:41 GMT
        Server: Werkzeug/0.9.4 Python/2.7.8

        {
            "success": true
        }

    """
    return jsonify(user="joe")


@app.route("/api/v1/users/create", methods=['POST'])
def create_user():
    """
        Since the path matches the regular expression r'/api/*', this resource
        automatically has CORS headers set.

        Browsers will first make a preflight request to verify that the resource
        allows cross-origin POSTs with a JSON Content-Type, which can be simulated
        as:
        $ curl --include -X OPTIONS http://127.0.0.1:5000/api/v1/users/create \
            --header Access-Control-Request-Method:POST \
            --header Access-Control-Request-Headers:Content-Type \
            --header Origin:www.examplesite.com
        >> HTTP/1.0 200 OK
        Content-Type: text/html; charset=utf-8
        Allow: POST, OPTIONS
        Access-Control-Allow-Origin: *
        Access-Control-Allow-Headers: Content-Type
        Access-Control-Allow-Methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT
        Content-Length: 0
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:22 GMT


        $ curl --include -X POST http://127.0.0.1:5000/api/v1/users/create \
            --header Content-Type:application/json \
            --header Origin:www.examplesite.com


        >> HTTP/1.0 200 OK
        Content-Type: application/json
        Content-Length: 21
        Access-Control-Allow-Origin: *
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:04 GMT

        {
          "success": true
        }

    """
    return jsonify(success=True)
```

(continues on next page)

(continued from previous page)

```
@app.route("/api/exception")
def get_exception():
    """
        Since the path matches the regular expression r'/api/*', this resource
        automatically has CORS headers set.

        Browsers will first make a preflight request to verify that the resource
        allows cross-origin POSTs with a JSON Content-Type, which can be simulated
        as:
        $ curl --include -X OPTIONS http://127.0.0.1:5000/api/exception \
            --header Access-Control-Request-Method:POST \
            --header Access-Control-Request-Headers:Content-Type \
            --header Origin:www.examplesite.com
        >> HTTP/1.0 200 OK
        Content-Type: text/html; charset=utf-8
        Allow: POST, OPTIONS
        Access-Control-Allow-Origin: *
        Access-Control-Allow-Headers: Content-Type
        Access-Control-Allow-Methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT
        Content-Length: 0
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:22 GMT
    """
    raise Exception("example")

@app.errorhandler(500)
def server_error(e):
    logging.exception('An error occurred during a request. %s', e)
    return "An internal error occured", 500


if __name__ == "__main__":
    app.run(debug=True)
```

Using the *cross_origins* decorator

```
@app.route("/", methods=['GET'])
@cross_origin()
def helloWorld():
    '''
        This view has CORS enabled for all domains, representing the simplest
        configuration of view-based decoration. The expected result is as
        follows:

        $ curl --include -X GET http://127.0.0.1:5000/ \
            --header Origin:www.examplesite.com

        >> HTTP/1.0 200 OK
        Content-Type: text/html; charset=utf-8
        Content-Length: 184
        Access-Control-Allow-Origin: *
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:29:56 GMT

        <h1>Hello CORS!</h1> Read about my spec at the
```

(continues on next page)

(continued from previous page)

```
        <a href="http://www.w3.org/TR/cors/">W3</a> Or, checkout my documentation
        on <a href="https://github.com/corydolphin/flask-cors">Github</a>

    '''
    return '''<h1>Hello CORS!</h1> Read about my spec at the
<a href="http://www.w3.org/TR/cors/">W3</a> Or, checkout my documentation
on <a href="https://github.com/corydolphin/flask-cors">Github</a>'''


@app.route("/api/v1/users/create", methods=['GET', 'POST'])
@cross_origin(allow_headers=['Content-Type'])
def cross_origin_json_post():
    '''
        This view has CORS enabled for all domains, and allows browsers
        to send the Content-Type header, allowing cross domain AJAX POST
        requests.

 Browsers will first make a preflight request to verify that the resource
        allows cross-origin POSTs with a JSON Content-Type, which can be simulated
        as:
        $ curl --include -X OPTIONS http://127.0.0.1:5000/api/v1/users/create \
            --header Access-Control-Request-Method:POST \
            --header Access-Control-Request-Headers:Content-Type \
            --header Origin:www.examplesite.com
        >> HTTP/1.0 200 OK
        Content-Type: text/html; charset=utf-8
        Allow: POST, OPTIONS
        Access-Control-Allow-Origin: *
        Access-Control-Allow-Headers: Content-Type
        Access-Control-Allow-Methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT
        Content-Length: 0
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:22 GMT


        $ curl --include -X POST http://127.0.0.1:5000/api/v1/users/create \
            --header Content-Type:application/json \
            --header Origin:www.examplesite.com


        >> HTTP/1.0 200 OK
        Content-Type: application/json
        Content-Length: 21
        Access-Control-Allow-Origin: *
        Server: Werkzeug/0.9.6 Python/2.7.9
        Date: Sat, 31 Jan 2015 22:25:04 GMT

        {
          "success": true
        }

    '''

    return jsonify(success=True)

if __name__ == "__main__":
    app.run(debug=True)
```

Index

C

CORS (*class in flask_cors*), 15
cross_origin() (*in module flask_cors*), 17