luhn Add CORS support for Refresh Token Grant. History

10 contributors

268 lines (222 sloc) | 10.7 KB

```python
"""
oauthlib.oauth2.rfc6749.grant_types
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
"""
import logging
from itertools import chain

from oauthlib.common import add_params_to_uri
from oauthlib.oauth2.rfc6749 import errors, utils
from oauthlib.uri_validate import is_absolute_uri

from ..request_validator import RequestValidator
from ..utils import is_secure_transport

log = logging.getLogger(__name__)


class ValidatorsContainer:
    """
    Container object for holding custom validator callables to be invoked
    as part of the grant type `validate_authorization_request()` or
    `validate_authorization_request()` methods on the various grant types.

    Authorization validators must be callables that take a request object and
    return a dict, which may contain items to be added to the `request_info`
    returned from the grant_type after validation.

    Token validators must be callables that take a request object and
    return None.
```

```python

    Both authorization validators and token validators may raise OAuth2
    exceptions if validation conditions fail.

    Authorization validators added to `pre_auth` will be run BEFORE
    the standard validations (but after the critical ones that raise
    fatal errors) as part of `validate_authorization_request()`

    Authorization validators added to `post_auth` will be run AFTER
    the standard validations as part of `validate_authorization_request()`

    Token validators added to `pre_token` will be run BEFORE
    the standard validations as part of `validate_token_request()`

    Token validators added to `post_token` will be run AFTER
    the standard validations as part of `validate_token_request()`

    For example:

    >>> def my_auth_validator(request):
    ...    return {'myval': True}
    >>> auth_code_grant = AuthorizationCodeGrant(request_validator)
    >>> auth_code_grant.custom_validators.pre_auth.append(my_auth_validator)
    >>> def my_token_validator(request):
    ...     if not request.everything_okay:
    ...         raise errors.OAuth2Error("uh-oh")
    >>> auth_code_grant.custom_validators.post_token.append(my_token_validator)
    """

    def __init__(self, post_auth, post_token,
                 pre_auth, pre_token):
        self.pre_auth = pre_auth
        self.post_auth = post_auth
        self.pre_token = pre_token
        self.post_token = post_token

    @property
    def all_pre(self):
        return chain(self.pre_auth, self.pre_token)

    @property
    def all_post(self):
        return chain(self.post_auth, self.post_token)


class GrantTypeBase:
    error_uri = None
    request_validator = None
    default_response_mode = 'fragment'
    refresh_token = True
    response_types = ['code']

    def __init__(self, request_validator=None, **kwargs):
        self.request_validator = request_validator or RequestValidator()

        # Transforms class variables into instance variables:
        self.response_types = self.response_types
        self.refresh_token = self.refresh_token
        self._setup_custom_validators(kwargs)
        self._code_modifiers = []
        self._token_modifiers = []

        for kw, val in kwargs.items():
            setattr(self, kw, val)

    def _setup_custom_validators(self, kwargs):
        post_auth = kwargs.get('post_auth', [])
        post_token = kwargs.get('post_token', [])
        pre_auth = kwargs.get('pre_auth', [])
        pre_token = kwargs.get('pre_token', [])
        if not hasattr(self, 'validate_authorization_request'):
            if post_auth or pre_auth:
                msg = ("{} does not support authorization validators. Use "
                       "token validators instead.").format(self.__class__.__name__)
                raise ValueError(msg)
            # Using tuples here because they can't be appended to:
            post_auth, pre_auth = (), ()
        self.custom_validators = ValidatorsContainer(post_auth, post_token,
                                                     pre_auth, pre_token)

    def register_response_type(self, response_type):
        self.response_types.append(response_type)

    def register_code_modifier(self, modifier):
        self._code_modifiers.append(modifier)

    def register_token_modifier(self, modifier):
        self._token_modifiers.append(modifier)

    def create_authorization_response(self, request, token_handler):
        """
        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        :param token_handler: A token handler instance, for example of type
                              oauthlib.oauth2.BearerToken.
        """
        raise NotImplementedError('Subclasses must implement this method.')

```

```python
    def create_token_response(self, request, token_handler):
        """
        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        :param token_handler: A token handler instance, for example of type
                              oauthlib.oauth2.BearerToken.
        """
        raise NotImplementedError('Subclasses must implement this method.')

    def add_token(self, token, token_handler, request):
        """
        :param token:
        :param token_handler: A token handler instance, for example of type
                              oauthlib.oauth2.BearerToken.
        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        """
        # Only add a hybrid access token on auth step if asked for
        if not request.response_type in ["token", "code token", "id_token token", "code id_token t
            return token

        token.update(token_handler.create_token(request, refresh_token=False))
        return token

    def validate_grant_type(self, request):
        """
        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        """
        client_id = getattr(request, 'client_id', None)
        if not self.request_validator.validate_grant_type(client_id,
                                                          request.grant_type, request.client, requ
            log.debug('Unauthorized from %r (%r) access to grant type %s.',
                      request.client_id, request.client, request.grant_type)
            raise errors.UnauthorizedClientError(request=request)

    def validate_scopes(self, request):
        """
        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        """
        if not request.scopes:
            request.scopes = utils.scope_to_list(request.scope) or utils.scope_to_list(
                self.request_validator.get_default_scopes(request.client_id, request))
        log.debug('Validating access to scopes %r for client %r (%r).',
                  request.scopes, request.client_id, request.client)
        if not self.request_validator.validate_scopes(request.client_id,
                                                      request.scopes, request.client, request):
            raise errors.InvalidScopeError(request=request)

    def prepare_authorization_response(self, request, token, headers, body, status):
        """Place token according to response mode.

        Base classes can define a default response mode for their authorization
        response by overriding the static `default_response_mode` member.

        :param request: OAuthlib request.
        :type request: oauthlib.common.Request
        :param token:
        :param headers:
        :param body:
        :param status:
        """
        request.response_mode = request.response_mode or self.default_response_mode

        if request.response_mode not in ('query', 'fragment'):
            log.debug('Overriding invalid response mode %s with %s',
                      request.response_mode, self.default_response_mode)
            request.response_mode = self.default_response_mode

        token_items = token.items()

        if request.response_type == 'none':
            state = token.get('state', None)
            if state:
                token_items = [('state', state)]
            else:
                token_items = []

        if request.response_mode == 'query':
            headers['Location'] = add_params_to_uri(
                request.redirect_uri, token_items, fragment=False)
            return headers, body, status

        if request.response_mode == 'fragment':
            headers['Location'] = add_params_to_uri(
                request.redirect_uri, token_items, fragment=True)
            return headers, body, status

        raise NotImplementedError(
            'Subclasses must set a valid default_response_mode')

    def _get_default_headers(self):
        """Create default headers for grant responses."""
        return {
            'Content-Type': 'application/json',
            'Cache-Control': 'no-store',
            'Pragma': 'no-cache',
        }

    def _handle_redirects(self, request):
        if request.redirect_uri is not None:
            request.using_default_redirect_uri = False
            log.debug('Using provided redirect_uri %s', request.redirect_uri)
            if not is_absolute_uri(request.redirect_uri):
                raise errors.InvalidRedirectURIError(request=request)

            # The authorization server MUST verify that the redirection URI
            # to which it will redirect the access token matches a
            # redirection URI registered by the client as described in
            # Section 3.1.2.
            # https://tools.ietf.org/html/rfc6749#section-3.1.2
            if not self.request_validator.validate_redirect_uri(
                    request.client_id, request.redirect_uri, request):
                raise errors.MismatchingRedirectURIError(request=request)
        else:
            request.redirect_uri = self.request_validator.get_default_redirect_uri(
                request.client_id, request)
            request.using_default_redirect_uri = True
            log.debug('Using default redirect_uri %s.', request.redirect_uri)
            if not request.redirect_uri:
                raise errors.MissingRedirectURIError(request=request)
            if not is_absolute_uri(request.redirect_uri):
                raise errors.InvalidRedirectURIError(request=request)

    def _create_cors_headers(self, request):
        """If CORS is allowed, create the appropriate headers."""
        if 'origin' not in request.headers:
            return {}

        origin = request.headers['origin']
        if not is_secure_transport(origin):
            log.debug('Origin "%s" is not HTTPS, CORS not allowed.', origin)
            return {}
        elif not self.request_validator.is_origin_allowed(
            request.client_id, origin, request):
            log.debug('Invalid origin "%s", CORS not allowed.', origin)
            return {}
        else:
            log.debug('Valid origin "%s", injecting CORS headers.', origin)
            return {'Access-Control-Allow-Origin': origin}
```