Test RTD Documentation

Release 1.0.0

Alberto Pagliarini

October 14, 2015

Contents

BEdita frontend app can be easily enabled to serve REST API. Once enabled the API present a set of default endpoints that can be customized for frontend needs.

CHAPTER 1

Setup a frontend to consume API

To use REST API in your frontend app you need at least BEdita 3.6.0 version. You can already also test it using 3-corylus branch.

Note: Because of authentication is handled using Json Web Token (IETF) and the JWT is digital signed using `'Security.salt'` you should always remember to change it in `app/config/core.php` file:

```
Configure::write('Security.salt', 'my-security-random-string');
```

1.1 Enable API on new frontend app

- from shell

```
cd /path/to/bedita
./cake.sh frontend init
```

- in `app/config/frontend.ini.php` define `$config['api']['baseUrl']` with your API base url, for example

```
$config['api'] = array('baseUrl' => '/api/v1');
```

That's all! You are ready to consume the API!

Point the browser to your API base url and you should see the list of endpoints available, for example

```
{
    "auth": "http://example.com/api/v1/auth",
    "me": "http://example.com/api/v1/me",
    "objects": "http://example.com/api/v1/objects",
    "poster": "http://example.com/api/v1/poster"
}
```

1.2 Enable API on old frontend app

- create a new `ApiController` in your frontend

```
require(BEDITA_CORE_PATH . DS . 'controllers' . DS . 'api_base_controller.php');

class ApiController extends ApiBaseController {
```

```
            //...
        }
```

- in `app/config/frontend.ini.php` define `$config['api']['baseUrl']` with your API base url.
- edit `app/config/routes.php` putting

```
        $apiBaseUrl = Configure::read('api.baseUrl');
        if (!empty($apiBaseUrl) && is_string($apiBaseUrl)) {
            Router::connect($apiBaseUrl . '/*', array('controller' => 'api', 'action' => 'route'));
        }
```

above

```
        Router::connect('/*', array('controller' => 'pages', 'action' => 'route'));
```

That's all!

After #570 we have implemented a new (and better) way to handle Exceptions. Remember to update your frontend `index.php` file:

```
if (isset($_GET['url']) && $_GET['url'] === 'favicon.ico') {
    return;
} else {
    $Dispatcher = new Dispatcher();
    $Dispatcher->dispatch();
}
```

Also make sure you have defined `views/errors/error.tpl` in your frontend for generic error handling.

CHAPTER 2

Configuration

To configure REST API you need to edit the frontend configuration file `app/config/frontend.ini.php`, for example

```
$config['api'] = array(
    'baseUrl' => '/api/v1',
    'allowedOrigins' => array(),
    'auth' => array(
        'component' => 'MyCustomAuth',
        'JWT' => array(
            'expiresIn' => 600,
            'alg' => 'HS256'
        ),
    ),
    'formatting' => array(
        'fields' => array(
            // fields that should be removed from results
            'remove' => array(
                'title',
                'Category' => array('name')
            ),
            // fields (removed by default) that should be kept
            'keep' => array(
                'ip_created',
                'Category' => array('object_type_id', 'priority')
            )
        )
    ),
    'validation' => array(
        'writableObjects' => array('document', 'event')
    )
);
```

Possible configuration params are:

- `baseUrl` the base url of REST API. Every request done to `baseUrl` will be handled as an API REST request via routing rules
- `allowedOrigins` define which origins are allowed. Leave empty to allow all origins
- `auth` contains authentication configurations:
- `component` define the name of auth component to use. By default `ApiAuth` Component is used
- `JWT` define some options used in Json Web Token authentication as the *"expires in"* time (in seconds) and the hashing algorithm to use

- `formatting` permits to setup some fomatting rules as object fields to *remove* or to *keep*
- `validation` setup some validation rules used generally in write operations. For example `writableObjects` define what object types are writable.

CHAPTER 3

Response and Errors

3.1 Response

Usually the response of API query has the structure

```
{
    "api": "objects",
    "data": {},
    "method": "get",
    "params": [],
    "url": "https://example.com/api/v1/objects/1"
}
```

where:

- `api` is the endpoint called
- `data` is an object containing all data requested
- `method` is the HTTP verb used in the request
- `params` contains all query url params used in the request
- `url` is the complete url requested (full base url + basepath + endpoint + other)

To set `data` for response is available the method `ApiBaseController::setData()` that accepts an array as first argument. A second argument permits to replace (default) or merge present data with that passed.

Other meta data can be placed inside response object, for example `paging` useful to paginate results:

```
{
    "api": "objects",
    "data": {},
    "method": "get",
    "paging": {
        "page": 1,
        "page_size": 10,
        "page_count": 10,
        "total": 995,
        "total_pages": 100
    },
    "params": [],
    "url": "https://example.com/api/v1/objects/1/children"
}
```

where:

- `page` is the current page
- `page_size` is the page dimension
- `page_count` is the number of items inside current page
- `total` if the count of all items
- `total_pages` is the total pages available

Note: If you need **to serve empty response body** to client you can use `ApiBaseController::emptyResponse()` that by default send a **204 No Content** HTTP status code. Pass another status code as first argument to send different status code.

3.2 Errors

Every time the API thrown an error the response will be similar to

```
{
    "error": {
        "status": 405,
        "code": null,
        "message": "Method Not Allowed",
        "details": "Method Not Allowed",
        "more_info": null,
        "url": "https://example.com/api/v1/foobar"
    }
}
```

where:

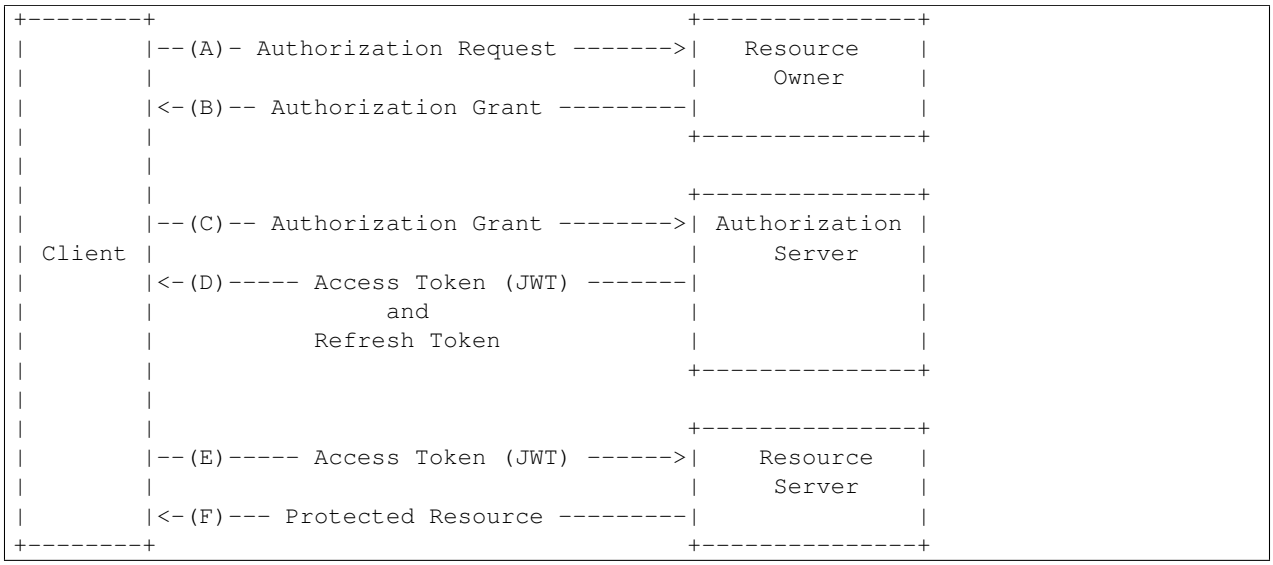
- `status` is the HTTP status code
- `code` is the API error code (not implemented)
- `message` is the error message
- `details` is the error detail
- `more_info` is the url to error documentation (not implemented)
- `url` is the url that has responded with the error

CHAPTER 4

Auhtentication

By default all GET requests don't require client and user authenticatication unless the object requested has permission on it. In that case the user has to be authenticated before require the resource. Other operations as writing/deleting objects (POST, PUT, DELETE on `/objects` endpoint) are always protected instead and they always require authentication.

The API follow a **token based authentication flow** using a Json Web Token as `access_token` and an opaque token as `refresh_token` useful to renew the `access_token` without ask again the user credentials. See here to more details on how to obtain `access_token` and `refresh_token`.



The `access_token` must be used in every request that require permission. To use the `access_token` it has to be sent in HTTP headers as **bearer token** `Authorization: Bearer eyJ0eXAi.....`

Note: Because of JWT is digital signed using `'Security.salt'` you should always remember to change it in `app/config/core.php` file:

```
Configure::write('Security.salt', 'my-security-random-string');
```

It is possible to invalidate all `access_token` released simply changing that value.

All the logic to handle authentication is in `ApiAuth` component and `ApiBaseController` use it for you so authentication works out of the box. If you need to protect *custom endpoints* you have to add to custom method

```
protected function customEndPoint() {
    if (!$this->ApiAuth->identify()) {
        throw new BeditaUnauthorizedException();
    }
}
```

4.1 Customize authentication

If you need to customize or change the authentication you can define your own auth component. To maintain the component method signature used in `ApiBaseController` your component should implements the interface `ApiAuthInterface`.

Remember that REST API are thought to implement token based authentication with the use of both `access_token` and `refresh_token` so the interface define methods to handle these tokens. If you need something different probably you would also override authenication methods of `ApiBaseController`.

In case you only need some little change it should be better to directly extend `ApiAuth` component that already implements the interface, and override the methods you need.

For example supposing you want to add additional check to user credentials, you can simply override `ApiAuth::authenticate()` method which deals with it:

```
App::import('Component', 'ApiAuth');

class CustomAuthComponent extends ApiAuthComponent {

    public function authenticate($username, $password, array $authGroupName = array()) {
        // authentication logic here
    }

}
```

and finally to activate the component all you have to do is define in configuration file `config/frontend.ini.php` the auth component you want to use.

```
$config['api'] = array(
    'baseUrl' => '/api',
    'auth' => array(
        'component' => 'CustomAuth'
    )
);
```

In `ApiController` you will have access to `CustomAuth` instance by `$this->ApiAuth` attribute.

CHAPTER 5

Pagination

Requesting a list of objects by `/objects` endpoint the result will be paginated using default values that you can *customize* in `ApiController`.

In the response you'll see tha pagination data in `paging` key

```
{
    "api": "objects",
    "data": {},
    "method": "get",
    "paging": {
        "page": 1,
        "page_size": 10,
        "page_count": 10,
        "total": 995,
        "total_pages": 100
    },
    "params": [],
    "url": "https://example.com/api/v1/objects/1/children"
}
```

where

- `page` is the current page
- `page_size` is the items per page
- `page_count` is the count of items in current page
- `total` is the total items
- `total_pages` is the total numbers of pages

To request a specific page simply call the endpoint passing `page` as GET parameter for example `/api/objects/1/children?page=5` to request the page 5.

You can also change the page size always through GET parameter, for example `/api/objects/1/children?page_size=50` to request 50 objects per page. `page_size` can't be greater of `$paginationOptions['maxPageSize']` defined in controller.

See below to know how to change the default values.

5.1 Define your API pagination default options

The default values used paginating items are defined in `ApiBaseController::paginationOptions` property.

```
protected $paginationOptions = array(
    'page' => 1,
    'pageSize' => 20,
    'maxPageSize' => 100
);
```

where `pageSize` is the default items per page and `maxPageSize` is the max page dimension that client can request. Requests with `page_size` greater of `maxPageSize` returns a 400 HTTP error.

If you want modify those defaults you can simply override that property in `ApiController`.

5.2 Paginate objects in custom endpoints

When a request has `page` or `page_size` as GET parameters those are validated and `$paginationOptions` is updated to contain the requested page options. A `dim` key equal to `pageSize` is added to be ready to use in some methods as `FrontendController::loadSectionObjects()`.

In this way in a 'custom' API endpoint you can simply do

```
protected function custom($id) {
    $result = $this->loadSectionObjects($id, $this->paginationOptions);
    // format and set pagination
    $this->setPaging($this->ApiFormatter->formatPaging($result['toolbar']));

    // do other stuff
}
```

and you are sure that pagination will work properly without doing anything else.

CHAPTER 6

Customize endpoints

6.1 Custom endpoints

Once you have enabled a frontend to consume API you have a set of default available endpoints visible pointing the browser to your API base url.

Sometimes you would want to define other endpoints to serve your custom data to clients. You can do it simply override the `$endpoints` attribute of `ApiBaseController`.

Write in your `ApiController`

```
protected $endPoints = array('friends');
```

and define the related custom method that will handle the data to show

```
protected function friends() {
    $friendsList = array('Tom', 'Jerry');
    $this->setData($friendsList);
}
```

The `setData()` method takes care of put `$friendsList` array inside response `data` key. Point the browser to your API base url you should see 'friends' in the endpoints list and if you request `GET /api/base/url/friends` you should see

```
{
    "api": "friends",
    "data": [
        "Tom",
        "Jerry"
    ],
    "method": "get",
    "params": [],
    "url": "https://example.com/api/v1/friends"
}
```

In this way all request types (GET, POST, PUT, DELETE) have to be handled by `friends()` method. Another possibility is to create one method for every request type allowed from the endpoint. It can be done creating methods named *"request type + endpoint camelized"*.

```
protected function getFriends() {
}

protected function postFriends() {
}
```

```
protected function putFriends() {
}

protected function deleteFriends() {
}
```

6.2 Blacklist endpoints

In some situations you will not want to expose some or all default endpoints, so in order to disable them it is possible to define a blacklist. After that calling those endpoints the response will be a **405 Method Not Allowed** HTTP error status code.

For example to blacklist "auth" and "objects" endpoints, in your `ApiController` override `$blacklistEndPoints` writing

```
protected $blacklistEndPoints = array('auth', 'objects');
```

Now, pointing to API base url you shouldn't see "auth" and "objects" endpoints anymore.

Pointing to them directly and you will receive a **405 HTTP error**.

6.3 Enable special object types endpoints

If you need you can also enable some special endpoint disabled by default. Those endpoints refer to BEdita object types mapping them to their pluralize form. So if you want to enable `/documents` end `/galleries` endpoints you have to edit `ApiController`

```
protected $whitelistObjectTypes = array('document', 'gallery');
```

These special endpoints automatically filter response objects through the object type related.

Again go to API base url to see 'documents' and 'galleries' added to endpoints list.

> **Note:** Note that those special endpoints work only for GET requests.

6.4 Customize `/objects` endpoint with your own URL path filter types

`objects` endpoint can be customized with URL path filters building endpoint structured as `/objects/:id/url_path_filter`. URL path filters on by default are visible in `ApiBaseController::$allowedObjectsUrlPath` property

```
protected $allowedObjectsUrlPath = array(
        'get' => array(
            'relations',
            'children',
            'contents',
            'sections',
            'descendants',
            'siblings',
            //'ancestors',
```

```
            //'parents'
        ),
        'post' => array(
            'relations',
            'children'
        ),
        'put' => array(
            'relations',
            'children'
        ),
        'delete' => array(
            'relations',
            'children'
        )
    );
```

URL path filters can be inhibited or new ones can be added overriding that property in `ApiController`.

In practice URL path filters are divided by request type (GET, POST, ...) so it is possible doing request like `GET /objects/1/children`, `POST /objects/1/relations` but not `POST /objects/1/siblings` because of that filter is active only for GET requests.

Every URL path filter must have a corresponding controller method named *"request type + Objects + URL path filter camelized"* that will handle the request. First url part *:id* and every other url parts after URL path filter will be passed to that method as arguments.

For example, supposing to want to remove all 'delete' and 'post' URL path filters and add a new 'foo_bar' filter for GET request, in `ApiController` we can override

```
protected $allowedObjectsUrlPath = array(
        'get' => array(
            'relations',
            'children',
            'contents',
            'sections',
            'descendants',
            'siblings',
            'foo_bar'
        ),
    );
```

and add the method

```
protected function getObjectsFooBar($objectId) {
    // handle request here
}
```

In this way the new URL path filter is active and reachable from `GET /objects/:id/foo_bar`. Every other request type (POST, PUT, DELETE) to that will receive **405 Method Not Allowed**.

If our 'foo_bar' URL path filter have to support `GET /objects/:id/foo_bar/:foo_val` requests then `ApiController::getObjectsFooBar()` will receive `:foo_val` as second argument. A best practice should be to add to method a validation on the number of arguments supported to avoid to respond to request as `GET /objects/:id/foo_bar/:foo_val/bla/bla/bla`.

```
protected function getObjectsFooBar($objectId, $fooVal = null) {
    if (func_num_args() > 2) {
        throw new BeditaBadRequestException();
    }
```

```
    // handle request here
}
```

CHAPTER 7

Formatting BEdita objects

7.1 Introducing the `ApiFormatter` Component

To respond with consistent data the BEdita object types are transformed and formatted using the `ApiFormatter` Component that deals with cleaning objects from useless data and casting and trasforming some fields in correct format.

If you have a look at `/objects/:id` response you'll see that fields as 'id' are **integer** other like 'latitude' and 'longitude' of geo tag are **float** and **dates are formatted in ISO-8601**. `ApiFormatter` Component with a little help from Models takes care of it.

When you load an object or list of objects you should always use the `ApiFromatter` Component to have data always formatted in the same way.

```
// load an object
$object = $this->loadObj($id);
$result = $this->ApiFormatter->formatObject($object);
// in $result['object'] you have the formatted object

// list of objects
$objects = $this->loadSectionObjects($id, array('itemsTogether' => true));
$result = $this->ApiFormatter->formatObjects($objects['children']);
// in $result['objects'] you have the formatted objects
```

`ApiFormatter::formatObject()` and `ApiFormatter::formatObjects()` accept as second argument an array of options with which it is possible add to the formatted object the count of relations and children.

```
$result = $this->ApiFormatter->formatObject($object, array(
    'countRelations' => true,
    'countChildren' => true
));
```

By default no count is done.

7.2 Help ApiFormatter to cast object fields in the right way

When formatting BEdita object `ApiFormatter` asks help to related object type Model to know which fields have to be cast in the right type. Basically every object type returns an array of fields that are defined in database as 'integer', 'float', 'date', 'datetime', 'boolean'. This array is returned from `BEAppObjectModel::apiTransformer()` method and it is something similar to

```
array(
    'id' => 'integer',
    'start_date' => 'datetime',
    'end_date' => 'datetime',
    'duration' => 'integer',
    'object_type_id' => 'integer',
    'created' => 'datetime',
    'modified' => 'datetime',
    'valid' => 'boolean',
    'user_created' => 'integer',
    'user_modified' => 'integer',
    'fixed' => 'boolean',
    'GeoTag' => array(
        'id' => 'integer',
        'object_id' => 'integer',
        'latitude' => 'float',
        'longitude' => 'float',
        'gmaps_lookat' => array(
            'latitude' => 'float',
            'longitude' => 'float',
            'zoom' => 'integer',
        )

    )
    'Tag' => array(
        'id' => 'integer',
        'area_id' => 'integer',
        'object_type_id' => 'integer',
        'priority' => 'integer',
        'parent_id' => 'integer',
    ),
    'Category' => array(
        'id' => 'integer',
        'area_id' => 'integer',
        'object_type_id' => 'integer',
        'priority' => 'integer',
        'parent_id' => 'integer',
    )
)
```

By default only tables that form the object chain plus 'categories', 'tags' and 'geo_tags' are automatically returned, but that method can be overridden to customize the result. For example the `Event` model add to basic transformer the `DateItem` transformer:

```
public function apiTransformer(array $options = array()) {
    $transformer = parent::apiTransformer($options);
    $transformer['DateItem'] = $this->DateItem->apiTransformer($options);
    return $transformer;
}
```

The `ApiFormatter` uses these transformers merged to common object transformer `ApiFormatterComponent::$transformers['object']` to present consistent data to client. It is possible to use some special transformer types that are:

- `underscoreField` that underscorize a camelcase field maintaining value unchanged
- `integerArray` that cast to integer all array values

7.3 Remove unwanted fields

Another useful task of `ApiFormatter` is to clean unwanted fields from data exposed to client. To do that it uses `ApiFormatter::$objectFieldsToRemove` array that can be customized through configuration or on the fly in controller.

7.3.1 Add fields to remove from configuration

In `config/frontend.ini.php` or `config/frontend.cfg.php` is possible to customize which fields exposed by default you want to remove from results.

```
$config['api'] = array(
    'baseUrl' => '/api/v1',
    ...
    'formatting' => array(
        'fields' => array(
            // fields that should be added
            // to ApiFormattingComponent::objectFieldsToRemove
            // i.e. removed from formatted object
            'remove' => array(
                'description',
                'title',
                'Category' => array('name'),
                'GeoTag' => array('title'),
                'Tag'
            )
        )
    )
);
```

In this way you say to `ApiFormatter` that 'description', 'title', 'name' of 'Category', 'title' of 'GeoTag' and all 'Tag' array must be cleaned from final results. Every time `ApiFormatter::formatObject()` or `ApiFormatter::formatObjects()` is called the data are cleaned up using `ApiFormatter::cleanObject()`.

7.3.2 Add fields to remove on the fly

In your `ApiController` you can decide in every moment to change which fields remove from results using `ApiFormatter::objectFieldsToRemove()` method.

```
// get the current value
$currentFieldsToRemove = $this->ApiFormatter->objectFieldsToRemove();

// to ovveride all. It completely replaces current fields to remove with new one
$this->ApiFormatter->objectFieldsToRemove(
    array(
        'title',
        'description'
    ),
    true
);

// to add new fields to remove
$this->ApiFormatter->objectFieldsToRemove(array(
```

```
    'remove' => array('title', 'description')
));
```

7.4 Keep fields that are removed by default

Sometime you could want to present to client some fields that normally are cleaned up. Likewise to what seen with fields to remove, it is possible do it from configuration or on the fly.

7.4.1 Add fields to keep from configuration

In `config/frontend.cfg.php`

```
$config['api'] = array(
    'baseUrl' => '/api/v1',
    ...
    'formatting' => array(
        'fields' => array(
            // fields that should be removed
            // to ApiFormattingComponent::objectFieldsToRemove
            // i.e. kept in formatted object
            'keep' => array(
                'fixed',
                'ip_created',
                'Category' => array('object_type_id', 'priority')
            )
        )
    )
);
```

In this way you say to `ApiFormatter` that 'fixed', 'ip_created' and 'object_type_id', 'priority' of 'Category' must be preserved and presented to client.

7.4.2 Add fields to keep on the fly

In your `ApiController`

```
// to keep fields
$this->ApiFormatter->objectFieldsToRemove(array(
    'keep' => array('ip_created', 'fixed')
));
```

It is possible to mix 'remove' and 'keep' options both in configuration and in controller.

CHAPTER 8

API reference

WARNING: This is a draft document, endpoints and data structure could likely change. We are still designing some parts of the response.

A frontend app enabled to consume REST API exposes a set of default endpoints:

8.1 Authentication

8.1.1 endpoint: /auth

It used to retrieve an `access_token` to access protected items, renew `access_token` and remove permissions. The `access_token` is a Json Web Token (IETF). More info on authentication

Note: Because of JWT is digital signed using `'Security.salt'` you should always remember to change it in `app/config/core.php` file:

```
Configure::write('Security.salt', 'my-security-random-string');
```

It is possible to invalidate all `access_token` released simply changing that value.

Obtain an access token

Request type: POST

Parameters

```
{
    "username": "test",
    "password": "test",
    "grant_type": "password"
}
```

`grant_type` is optional because it is the default used if no one is passed.

If user is validated the response will contain the JWT, the time to expire (in seconds) and the refresh_token useful to renew the JWT

```
{
    "api": "auth",
    "data": {
        "access_token": "eyJ0eXAi.....",
```

```
        "expires_in": 600,
        "refresh_token": "51a3f718e7abc712e421f64ea497a323aea4e76f"
        },
    "method": "post",
    "params": [ ],
    "url": "https://example.com/api/auth"
}
```

Once you received the access token you have to use it in every request that require authentication. It can be used in HTTP header

```
Authorization: Bearer eyJ0eXAi.....
```

or as query url `/api/endpoint?access_token=eyJ0eXAi.....`

Renew the access token

If the access token was expired you need to generate a new one started by refresh token. **In this case do not pass the expired access token**

Request type: POST

Parameters

```
{
    "grant_type": "refresh_token",
    "refresh_token": "51a3f718e7abc712e421f64ea497a323aea4e76f"
}
```

If refresh token is valid it returns the new access token

```
{
    "api": "auth",
    "data": {
        "access_token": "rftJasd3.....",
        "expires_in": 600,
        "refresh_token": "51a3f718e7abc712e421f64ea497a323aea4e76f"
        },
    "method": "post",
    "params": [ ],
    "url": "https://example.com/api/auth"
}
```

Get the updated time to access token expiration

Calling /auth in GET using the access_token return the updated 'expires_in' time.

Request type: GET

It returns

```
{
    "api": "auth",
    "data": {
        "access_token": "rftJasd3.....",
        "expires_in": 48
    },
    "method": "get",
```

```
    "params": [ ],
    "url": "https://example.com/api/auth"
}
```

Revoking a refresh token /auth/:refresh_token

In order to invalidate an access_token you need to remove it from client and revoke the refresh token

Request type: DELETE

If the refresh token is deleted it responds as HTTP 204 No Content.

8.2 Objects

8.2.1 endpoint: /objects

It used to get a BEdita object or a set of objects.

Get an object /objects/:name

where *:name* **can be the object id or the object unique name (nickname)**. Note that the response data fields can change depending of BEdita object type exposed so you could see more or less fields respect to example below.

Every object can have relations with other objects. The count of objects related is in `data.object.relations.<relation_name>` where there are `count` (the number of related object) and `url` fields. The `url` is the complete API request url to get the object related, for example https://example.com/api/objects/1/relations/attach

Request type: GET

```
{
    "api": "objects",
    "data": {
        "object": {
            "id": 218932,
            "start_date": "2015-01-08T00:00:00+0100",
            "end_date": null,
            "subject": null,
            "abstract": null,
            "body": "This is the body text",
            "object_type_id": 22,
            "created": "2015-01-30T10:04:49+0100",
            "modified": "2015-05-08T12:59:49+0200",
            "title": "hello world",
            "nickname": "hello-world",
            "description": "the description",
            "valid": true,
            "lang": "eng",
            "rights": "",
            "license": "",
            "creator": "",
            "publisher": "",
            "note": null,
            "comments": "off",
```

```
            "publication_date": "2015-01-08T00:00:00+0100",
            "languages": {
                "ita": {
                    "title": "ciao mondo"
                }
            },
            "relations": {
                "attach": {
                    "count": 8,
                    "url": "https://example.com/api/objects/1/relation/attach"
                },
                "seealso": {
                    "count": 2,
                    "url": "https://example.com/api/objects/1/relation/seealso"
                }
            },
            "object_type": "Document",
            "authorized": true,
            "free_access": true,
            "custom_properties": {
                "bookpagenumber": "12",
                "number": "8"
            },
            "geo_tags": [
                {
                    "id": 68799,
                    "object_id": 218932,
                    "latitude": 44.4948179,
                    "longitude": 11.33969,
                    "address": "via Rismondo 2, Bologna",
                    "gmaps_lookats": {
                        "zoom": 16,
                        "mapType": "k",
                        "latitude": 44.4948179,
                        "longitude": 11.33969,
                        "range": 44052.931589613
                    }
                }
            ],
            "tags": [
                {
                    "label": "tag one",
                    "name": "tag-one"
                },
                {
                    "label": "tag two",
                    "name": "tag-two"
                }
            ],
            "categories": [
                {
                    "id": 266,
                    "area_id": null,
                    "label": "category one",
                    "name": "category-one"
                },
                {
                    "id": 323,
```

```
                    "area_id": null,
                    "label": "category two",
                    "name": "category-two"
                }
            ]
        }
    },
    "method": "get",
    "params": [],
    "url": "https://example.com/api/objects/218932"
}
```

If *:name* corresponds to a **section** or a **publication** then the response will have `data.object.children` with the total count of children, count of contents, count of sections and the related url.

```
{
    "children": {
        "count": 14,
        "url": "https://example.com/api/objects/1/children",
        "contents": {
            "count": 12,
            "url": "https://example.com/api/objects/1/contents"
        },
        "sections": {
            "count": 2,
            "url": "https://example.com/api/objects/1/sections"
        }
    }
}
```

Get a list of publication's descendants /objects

Request type: GET

Return a paginated list of objects that are descendants of the frontend publication. The response will be an array of objects as shown below.

Get a list of related objects /objects/:name/:filter_type

Return a list of objects related to *:name* object using *:filter_type* filter.

:filter_type value can be 'ancestors' (not supported yet), 'parents' (not supported yet), 'children', 'descendants', 'siblings', 'contents', 'sections' and 'relations'

The response will usually be an array of objects as:

Request type: GET

```
{
    "api": "objects",
    "data": {
        "objects": [
            {
                "id": 100,
                "title": "my title",
                ...
            },
```

```
            {
                "id": 42,
                "title": "other title",
                ...
            },
            ...
        ]
    },
    "method": "get",
    "paging": {
        "page": 1,
        "page_size": 5,
        "page_count": 5,
        "total": 50,
        "total_pages": 10
    },
    "params": [],
    "url": "https://example.com/api/objects/1/children"
}
```

Get a list of children /objects/:name/children

It returns the paginated children of object *:name*.

Get a list of children of type section /objects/:name/sections

It returns the paginated children of object *:name*. The children are just sections ('section BEdita object type)

Get a list of children of type contents /objects/:name/contents

It returns the paginated children of object *:name*. The children are other than sections.

Get a list of descendants /objects/:name/descendants

It returns the paginated descendants of object *:name*.

Get a list of siblings /objects/:name/siblings

It returns the paginated siblings of object *:name*.

Get relations count /objects/:name/relations

It returns a summary of relations information about *:name* object. It show every relation with the count and the url to get the related objects detail.

```
{
    "api": "objects",
    "data": {
        "attach": {
            "count": 1,
```

```
            "url": "https://example.com/api/objects/1/relations/attach"
        },
        "seealso": {
            "count": 2,
            "url": "https://example.com/api/objects/1/relations/seealso"
        }
    },
    "method": "get",
    "params": [],
    "url": "https://example.com/api/objects/1/relations"
}
```

Get the related objects detail /objects/:name/relations/:relation_name

It returns the paginated list of objects related by *:relation_name* to *:name* object.

Get the relation detail /objects/:name/relations/:relation_name/:related_id

Request type: GET

It returns the relation *:relation_name* detail from main object *:name* and related object *related_id*

```
{
  "api": "objects",
  "data": {
    "priority": 3,
    "params": {
      "label": "here the label"
    }
  },
  "method": "get",
  "params": [],
  "url": "https://example.com/api/objects/1/relations/attach/2"
}
```

Get the child position /objects/:name/children/:child_id

Request type: GET

It returns the position (`priority` key) of *:child_id* relative to all children of parent object *:name*

```
{
  "api": "objects",
  "data": {
    "priority": 3
  },
  "method": "get",
  "params": [],
  "url": "https://example.com/api/objects/1/children/2"
}
```

8.2.2 Create/update an object

Request type: POST

Conditions: User has to be authenticated and has to have the permission to access to the object.

Before save objects the frontend app that serve API has to be configured to know what objects can be written

```
$config['api'] = array(
   ....
    'validation' => array(
        // to save 'document' and 'event' object types
        'writableObjects' => array('document', 'event')
    )
);
```

Saving new objects user has to be authenticated and **data** from client must contain: * `object_type` i.e. the object type you want to create * at least a parent (`parents` key) accessible (with right permission for user authorized) on publication tree or at least a relation (`relations` key) with another object reachable (where *reachable* means an accessible object on tree or related to an accessible object on tree).

Example of valid data from client:

```
{
    "data": {
        "title": "My title",
        "object_type": "event",
        "description": "bla bla bla",
        "parents": [1, 34, 65],
        "relations": {
            "attach": [
                {
                    "related_id": 12,
                    "params": {
                        "label": "foobar"
                    }
                },
                {
                    "related_id": 23
                }
            ],
            "seealso": [
                {
                    "related_id": 167
                }
            ]
        },
        "categories": ["name-category-one", "name-category-two"],
        "tags": ["name-tag_one", "name-tag-two"],
        "geo_tags": [
            {
                "title": "geo tag title",
                "address": "via ....",
                "latitude": 43.012,
                "longitude": 10.45
            }
        ],
        "date_items": [
            {
                "start_date": "2015-07-08T15:00:35+0200",
                "end_date": "2015-07-08T15:00:35+0200",
                "days": [0,3,4]
            },
            {
```

```
                "start_date": "2015-09-01T15:00:35+0200",
                "end_date": "2015-09-30T15:00:35+0200"
            }
        ]
    }
}
```

dates must be in ISO 8601 format. In case of **success** a **201 Created** HTTP status code is returned with the detail of object created in the response body.

You can use POST also to **update an existent object**. In that case the object `id` has to be passed in "data" object from client and `object_type` can be omitted.

8.2.3 Add/edit relations

Request type: POST

Conditions: User has to be authenticated and has to have the permission to access to the object.

In order to add or edit relations you can use the endpoint `/objects` as `/objects/:name/relations/:relation_name` where *:name* can be the object id or nickname. and *:relation_name* the relation name. Relations data must be an array of relation data or an object with relation data if you need to save only one relation (note that it is the same that send an array with only one relation).

- `related_id` is the related object id and is mandatory
- `params` fields depend from relation type (optional)
- `priority`is the position of the relation. Relation with lower priority are shown before (optional)

For example to add/edit attach relations to object with id 3 you can do a request:

`POST /objects/3/relations/attach`

valid data can be:

```
{
    "data": [
        {
            "related_id": 15,
            "params": {
                "label": "my label"
            }
        },
        {
            "related_id": 28
        }
    ]
}
```

to create/update a bunch of relations, or

```
{
    "data": {
        "related_id": 34,
        "priority": 3
    }
}
```

to create/update only one relation.

If a "relation_name" relation between main object and related object not exists then it is created else it is updated. If at least a relation is created a **201 Created** HTTP status code is sent and an HTTP header **Location** is set with url of *list of related objects*.

The response body will be an array of relation data just saved.

Saving new relations you can pass the `priority` you want to set. If no `priority` is passed it is automatically calculated starting from the max `priority` in the current relation.

8.2.4 Edit (replace) relation data between objects

Request type: PUT

Conditions: User has to be authenticated and has to have the permission to access to the objects.

In order to edit the relation data between two objects you can use the endpoint `/objects` as `/objects/:name/relations/:relation_name/:related_id` where *:name* can be the object id or nickname, *:relation_name* the relation name and *:related_id* the related object id. Relations data must be an object with data

- `params` fields depend from relation type
- `priority` is the position of the relation. Relation with lower priority are shown before

At least `params` or `priority` must be defined. If one of these is not passed it will be set to `null`.

So to edit attach relation between object 1 and 2 the request will be

```
PUT /objects/1/relations/attach/2
```

```
{
    "data": {
        "priority": 3,
        "params": {
            "label": "new label"
        }
    }
}
```

In case of success the server will respond with a **200 HTTP status code** and the response body will be the same of *Get the relation detail*

8.2.5 Add/edit children

Request type: POST

Conditions: User has to be authenticated and has to have the permission to access to the object.

In order to add or edit children to a area/section object type you can use the endpoint `/objects` as `/objects/:name/children` where *:name* can be the object id or nickname. Children data must be an array of child data or an object with child data if you need to save only one child (note that it is the same that send an array with only one child).

- `child_id` is the child object id and is mandatory
- `priority` is the position of the child on the tree

For example to add/edit children to object with id 3 you can do a request:

```
POST /objects/3/children
```

valid data can be:

```
{
    "data": [
        {
            "child_id": 15,
            "priority": 3
        },
        {
            "child_id": 28
        }
    ]
}
```

to create/update a bunch of children, or

```
{
    "data": {
        "child_id": 34,
        "priority": 3
    }
}
```

to create/update only one child.

If a "child_id" is a new children for parent object then it is created on tree else it is updated. If at least a new child is created a **201 Created** HTTP status code is sent and an HTTP header **Location** is set with url of *list of children objects*.

The response body will be an array of children data just saved.

Saving new children you can pass the `priority` you want to set i.e. the position on the tree. If no `priority` is passed every new children is appended to parent on tree structure.

8.2.6 Edit children position

Request type: PUT

Conditions: User has to be authenticated and has to have the permission to access to the objects.

In order to edit children position you can use the endpoint `/objects` as `/objects/:name/children/:child_id` where *:name* can be the object id or nickname and *:child_id* is the children object id. Data passed must contain `priority` field that is the position of child you want to update.

For example to edit the position of child with id 2 of parent with id 1:

`PUT /objects/1/children/2`

```
{
    "data": {
        "priority": 5
    }
}
```

8.2.7 Delete an object

Request type: DELETE

Conditions: User has to be authenticated and has to have the permission to access to the object.

To delete an object has to be used the endpoint `/objects/:name` where *:name* can be the object id or nickname.

If the object is deleted successfully a **204 No Content** HTTP status code is sent. Further requests to delete the same object will return a **404 Not Found** HTTP status code.

8.2.8 Delete a relation between objects

Request type: DELETE

Conditions: User has to be authenticated and has to have the permission to access to the object.

In order to delete an existent relation between two objects you can use the endpoint `/objects/:name/relations/:rel_name/:related_id` where *:name* is the object id or nickname, *:rel_name* is the relation name between objects and *:related_id* is the object id related to object *:name*.

If the relation is succesfully deleted *204 No Content* HTTP status code is sent. Further requests to delete the same relation will return a **404 Not Found** HTTP status code.

8.2.9 Remove child from a parent

Request type: DELETE

Conditions: User has to be authenticated and has to have the permission to access to the object.

To remove an existent child of an object the endpoint `/objects/:name/children/:child_id` can be used, where *:name* is the object id or nickname of parent and *:child_id* is id of the child object. Note that the child will be only removed from parent's tree but it continue to exist.

If *:child_id* is succesfully removed from *:name* children a **204 No Content** HTTP status code is sent. Further requests to remove the same child will return a **404 Not Found** HTTP status code.

8.3 Poster

8.3.1 endpoint: /poster/:id

Get the image representation of object *:id* as thumbnail url

Request type: GET

Return the thumbnail url of an image representation of the object *:id*. The thumbnail returned depends from the object type of *:id* and from its relations, in particular:

1. if object *:id* has a 'poster' relation with an image object then it returns a thumbnail of that image 2.else if the object is an image object then it returns a thumbnail of the object
2. else if the object has an 'attach' relation with an image object then it returns a thumbnail of that image

The response will be

```
{
  "api": "poster",
  "data": {
    "id": 5,
    "uri": "http://media.server/path/to/thumb/thumbnail.jpg"
  },
  "method": "get",
```

```
  "params": [],
  "url": "https://example.com/api/poster/5"
}
```

Query Url Parameters

- `width` the thumbnail width
- `height` the thumbnail height

8.4 User profile

8.4.1 endpoint: /me

Obtain information about authenticated user

Request type: GET

Conditions: User has to be authenticated

Return information about current authenticated user

CHAPTER 9

Indices and tables

- genindex
- modindex
- search