BEdita Documentation

Release 3.8.0

Channelweb Srl, Chialab Srl

May 19, 2017

REST API

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');
```

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": "https://example.com/api/v1/auth",
    "me": "https://example.com/api/v1/me",
    "objects": "https://example.com/api/v1/objects",
    "posters": "https://example.com/api/v1/posters"
}
```

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)) {
    $apiBaseUrl .= (substr($apiBaseUrl, -1) === '/') ? '*' : '/*';
    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'),
        'allowedUrlParams' => array(
           'endpoint_name' => array('param_one', 'param_two')
        )
    ),
    'upload' => array(
        'quota' => array(
```

```
            'maxFileSize' => 8*1024*1024, // 8 MB
            'maxSizeAvailable' => 50*1024*1024, // 50 MB
            'maxFilesAllowed' => 500
        )
    )
);
```

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 or to validate request and data:
  - `writableObjects` define what object types are writable
  - `allowedUrlParams` define which *query string paramters are valid for endpoints*
- `upload` the upload configuration. Contains:
  - `quota` is an array with space and files limits available for every user. See *Limiting max file size allowed and quota available*.

CHAPTER 3

Response and Errors

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.

`data` fields format

Every response in data is formatted in the right type usually according to database fields type. So it can contain values as integers, floats, booleans and dates.

While other types are clear we need to clarify the format used for dates. All reponse dates are formatted using ISO-8601 standard combining date and time, for example `2016-11-14T15:54:01+01:00`.

The client should always send dates in that format or using the form with **Z** as zone designator to indicate UTC, coming for example from javascript function `Date().toISOString()`.

So the same date should be sent from client as `2016-11-14T15:54:01+01:00` or `2016-11-14T14:54:01.640Z`.

Tip: If you need to customize the format data types you could be interested in reading *Help ApiFormatter to cast object fields in the right way*

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 (if exists)
- `message` is the error message
- `details` is the error detail
- `more_info` can contain useful information for better understand the error.
- `url` is the url that has responded with the error

Error codes

Here is a list of the error codes thrown from the API.

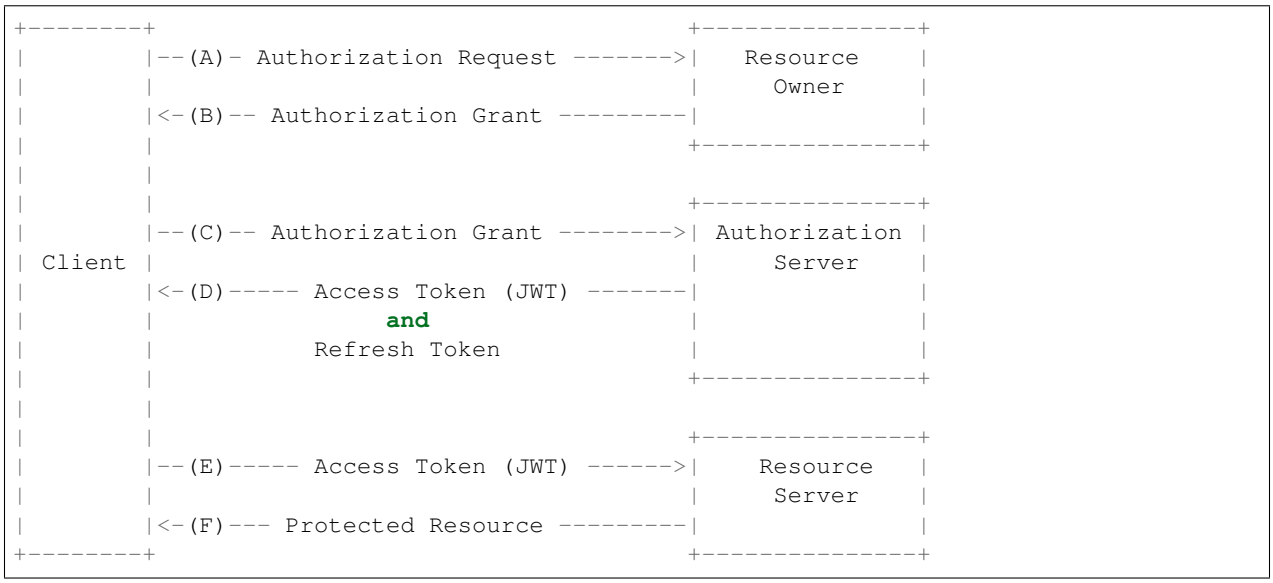
Error Code	Description	HTTP status code
UPLOAD_MAX_FILESIZE_EXCEEDED	Upload max file size exceeded	400 Bad Request
UPLOAD_QUOTA_EXCEEDED	Upload quota available exceeded	403 Forbidden
UPLOAD_FILES_LIMIT_EXCEEDED	Maximum number of files allowed exceeded	403 Forbidden

CHAPTER 4

Authentication

Architecture

The API follow a **token based authentication flow** using a *JSON Web Token* as *access token* and an opaque token as *refresh token*.



Usually *JWT* payload contains the user `id` and some public claims, for example

```
{
    "iss": "https://example.com",
    "iat": "1441749523",
    "exp": "1441707000",
    "id": "15"
}
```

Important: 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.

By default all GET requests don't require client and user authentication 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.

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();
    }
}
```

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.

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`.

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

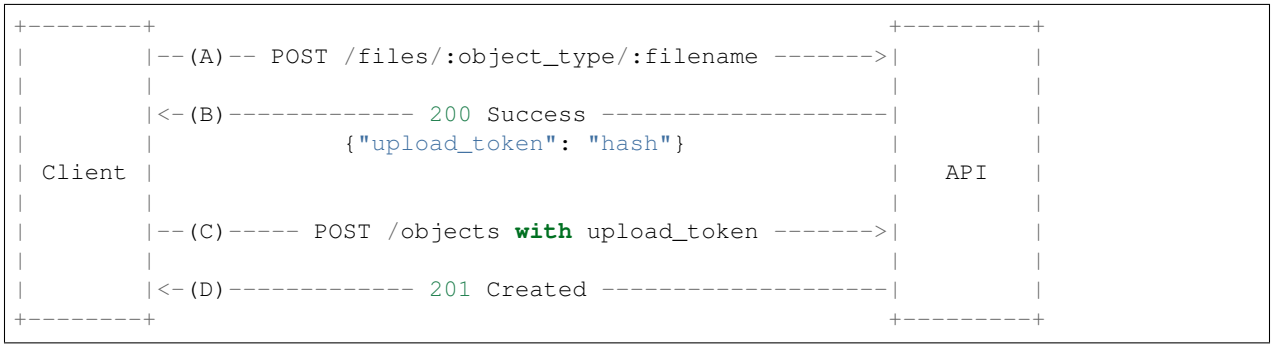
```
protected function custom($id) {
    $result = $this->loadSectionObjects($id, $this->paginationOptions);
    // format and set pagination
    $this->setPaging($this->ApiFormatter->formatPaging($result['toolbar']));

    // do other stuff
}
```

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

CHAPTER 6

Uploading files

The BEdita REST API allows to upload files and link it to BEdita objects. The process of upload follow the schema:



where:

1. The client send a request to upload a file named *:filename* related to an object type *:object_type*. See *POST /files/(string:object_type)/(string:file_name)* for more info.
2. The API responds with 200 OK and an *upload token*.
3. The client finalize the upload request creating a BEdita object and using the *upload token* received for link the new object to the file uploaded
4. The API responds with 201 Created as shown in *POST /objects*

Limiting which mime types are allowed

If you want to limit the mime types of files uploadable you can edit the frontend configuration in `app/config/frontend.cfg.php` defining for every object type an array of regexp used to validate the mime type. If no one is defined the rules that you can find in `bedita-app/config/bedita.ini.php` will be applied.

An example of configuration could be:

```
$config['validate_resource']['mime']['Image'] = array('/image\/png/', '/image\/gif/');
$config['validate_resource']['mime']['Video'] = array('/video\/\.*/');
```

To allow `image/png` and `image/gif` mime types for `Image` object and all `video/*` mime types for `Video` object.

Limiting max file size allowed and quota available

Usually you would want to limit the amount of files and space available for every user. The default max file size supported and the quota available are defined in `ApiUploadComponent` as

```
protected $quota = array(
    'maxFileSize' => 52428800,  // max file size uploadable 50 MB
    'maxSizeAvailable' => 524288000,  // quota available 500 MB
    'maxFilesAllowed' => 500 // max number of files uploadable
);
```

These parameters are configurable editing `app/config/frontend.cfg.php`, for example

```
$config['api']['upload'] => array(
    'quota' => array(
        'maxFileSize' => 8*1024*1024, // 8 MB
        'maxSizeAvailable' => 50*1024*1024, // 50 MB
        'maxFilesAllowed' => 100
    )
);
```

Creating a custom object type supporting upload

If you want to use `/files` endpoint for your custom object types, your model must follow some conventions:

- it can **extends** `BeditaSimpleStreamModel` or `BeditaStreamModel` as done by core multimedia `object_type` (image, video, audio, b_e_file, application).

  In this case the upload will follow the classic BEdita flow putting the file uploaded in the path defined in configuration as `$config['mediaRoot']` and reachable from `$config['mediaUrl']` url unless you follow the below convention.

- if you need to handle the upload in a way different, the object model must **implements**:

  interface `UploadableInterface`

  `apiUpload`(*File $source*, *array $options*)
  Used for execute the upload action given the source file. It deals with moving the uploaded file in a target destination returning the target url.

  Parameters
  - **`$source`** (*`File`*) – The instance of CakePHP File referred to the file uploaded
  - **`$options`** (*`array`*) – An array of options containing:
    - `fileName`: the safe file name obtained from source original file name
    - `hashFile`: the md5 hash of the source file
    - `user`: the user data identified by *access token*

  Returns Either false on failure, or the target url where the the file is reachable.

apiUploadTransformData (*array $uploadData*)
Used for finalize the upload action adding additional fields to object data to save.

Parameters

- **$uploadData** (*array*) – An array of data containing information about file previously uploaded:
  - `uri`: the url of file uploaded (that one returned from `apiUpload()`)
  - `name`: the safe file name
  - `mime_type`: the file mime-type
  - `file_size`: the file size
  - `original_name`: the original file name uploaded
  - `hash_file`: the md5 hash of file

Returns array of additional fields to add to object data passed in creation according to `Model` table used.

apiUploadQuota (*array $uploadableObjects*, *array $user*, *$event*)
It should return the quota usage and total amount of uploaded objects for `$user`. This method is intended to be bound to `Api.uploadQuota` event, so the model should add it as listener.

For example:

```
class CustomObjectType {

    public function __construct() {
        parent::__construct();
        BeLib::eventManager()->bind(
            'Api.uploadQuota',
            array('CustomObjectType', 'apiUploadQuota')
        );
    }

    public function apiUploadQuota(array $uploadableObjects, array
↪$user, $event) {
        // do stuff to calculate object type quota used
        return array_merge($event->result, $calculatedQuota);
    }

}
```

Parameters

- **$uploadableObjects** (*array*) – Array of uploadable object types
- **$user** (*array*) – User data
- **$event** – The dispatched event

Returns

array or false in case of error.

The result must be merged with `$event->result` and returned in the form:

```
array(
    'object_type_name' => array(
        'size' => 12345678,
```

```
        'number' => 256
    ),
)
```

apiCreateThumbnail(*$id*, *$thumbConf = array()*)

Create the thumbnail for the requested resource (if necessary) and returns an array containing the uri of the thumb and the object id refered from thumb.

Parameters

- **$id** (*int*) – The object id
- **$thumbConf** (*array*) – The thumbnail configuration

Returns

array or false

It must return `false` if the thumb creation fails else the array must contain the keys `id` and `uri`

```
array(
    'id' => 15,
    'uri' => 'https://example.com/thumb/thumb_name.jpg'
)
```

CHAPTER 7

Customize endpoints

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() {
}
```

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**.

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.

Setup query string parameters

You can easily customize which query string parameters an endpoint can accept. By deafult every endpoint accepts the *access_token* query string (also if it is suggested to pass it in HTTP header Authorization). That is also valid for custom endpoints you create.

Moreover *default endpoints* support additional query string params according to `ApiBaseController::$defaultAllowedUrlParams`.

Every time a request is fullfilled with query strings parameters they are validated against those allowed. If validation fails the response return a 400 Bad Request status code.

To validate your own query string parameters there are two ways. Directly in `ApiController` or via configuration.

Configure allowed query string parameter in `ApiController`

To validate your own query string parameters you can define the `$allowedUrlParams` in `ApiController` as

```
protected $allowedUrlParams = array(
    'endpoint_name' => array('param_one', 'param_two')
);
```

Then you can send request as

```
GET /endpoint_name?param_one=value HTTP/1.1
```

without receive a 400 Bad Request error.

To group some parameters that you want to make availbale to more endpoints, the _ prefix can be used in the array keys. It will be considered as a special word used as *group name* instead of *endpoint name*.

```
protected $allowedUrlParams = array(
    // define a group of params
    '_paramGroup' => array('param_one', 'param_two'),
    // use group in endpoints
    'endpoint_one' => array('_paramGroup'),
    'endpoint_two' => array('_paramGroup', 'param_three')
);
```

Configure allowed query string parameter in configuration file

Using the same convention seen *above* you can customize the allowed query string editing `app/config/frontend.ini.php` or `app/config/frontend.cfg.php`

```
$config['api'] = array(
    'baseUrl' => '/api',
    // other conf params
    // ...
    'validation' => array(
        'allowedUrlParams' => array(
            // define a group of params
            '_paramGroup' => array('param_one', 'param_two'),
            // use group in endpoints
            'endpoint_one' => array('_paramGroup'),
            'endpoint_two' => array('_paramGroup', 'param_three')
        )
    )
);
```

Add allowed query string parameters on the fly

If you need to change the allowed parameters on the fly `ApiValidator` provides a method to register them.

```
// In ApiController
$allowedParams = array('objects' => 'custom_param');
// add "custom_param" to "objects" endpoint allowed params
$this->ApiValidator->registerAllowedUrlParams($allowedParams);

// or if you want override all rules with new one
$this->ApiValidator->registerAllowedUrlParams($allowedParams, false);
```

Customize `/objects` endpoint

Here we'll see how to customize the *objects* endpoint.

Configure 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
}
```

Configure query string paramters to filter objects

Previoulsy we have seen how to add custom allowed query string params to endpoints.

The default allowed params are visible in *`GET /objects`*. In particular a special query string parameter is used to filter objects, its name is `filter[]` and it's an array of conditions to apply to get collections of objects.

For example

```
GET /objects?filter[object_type]=document,gallery,event HTTP/1.1
```

will return a collection of publication's descendants of type *document* or *gallery* or *event*. Filters are chained so you can do

```
GET /objects?filter[object_type]=document,gallery&filter[query]=test HTTP/1.1
```

to obtain a collection of publication's descendants of type *document* or *gallery* containing the word "test" in some of their indexed fields.

In general you can define other useful filters with this convention

```
filter[objects_table_field_name]
```

where "objects_table_field_name" is a field of the objects table.

Or

```
filter[Model.field_name]
```

if you want to filter on a field in another table that extends objects table. For example if you want to filter by *name* field in *cards* table we would configure the allowed param `filter[Card.name]`

```
// frontend.ini.php
$config['api'] = array(
    'baseUrl' => '/api',
    // other conf params
    // ...
    'validation' => array(
        'allowedUrlParams' => array(
            'objects' => array('filter[Card.name]')
        )
    )
);
```

then we can search all publication's descendants of type *card* with *name* equal to "Tom" or "Jerry".

```
GET /objects?filter[object_type]=card&filter[Card.name]=Tom,Jerry HTTP/1.1
```

CHAPTER 8

Formatting BEdita objects

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.

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

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.

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()`.

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')
));
```

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.

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.

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 9

API reference

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

Note: Every POST request can send the payload as `x-www-form-urlencoded` or `application/json`. For readability all examples will use `Content-type: application/json`.

Authentication `/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*

Important: 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

`POST /auth`

Authenticate an user to obtain an *access token*.

Request JSON Object

- **`username`** (*string*) – the username
- **`password`** (*string*) – the password

- **grant_type** (*string*) – *"password"*, the grant type to apply (password is the default, it can be ommitted)

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – response contains *access token* and *refresh token*
- 400 Bad Request – when required parameters are missing or the request is malformed
- 401 Unauthorized – when authentication fails

Example request:

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "api": "auth",
    "data": {
        "access_token": "eyJ0eXAi.....",
        "expires_in": 600,
        "refresh_token": "51a3f718e7abc712e421f64ea497a323aea4e76f"
        },
    "method": "post",
    "params": [ ],
    "url": "https://example.com/api/auth"
}
```

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

```
Authorization: Bearer <token>
```

or as query string `/api/endpoint?access_token=<token>`

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**

POST /auth

Renew an *access_token*.

Request JSON Object

- **refresh_token** (*string*) – the *refresh token* to use to renew *access token*
- **grant_type** (*string*) – *"refresh_token"*, the grant type to apply

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success, it responds with the new *access token* and *refresh token*
- 400 Bad Request – when required parameters are missing or the request is malformed
- 401 Unauthorized – when *refresh token* is invalid

Example request:

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "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

GET /auth

It returns the updated `expires_in` time for *access token*

Request Headers

- Authorization – the *access token* as Bearer token

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – no error, payload contains the updated `expires_in` value
- 400 Bad Request – the request is malformed
- 401 Unauthorized – the *access token* is invalid

Example request:

```
GET /auth HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "api": "auth",
    "data": {
        "access_token": "rftJasd3.....",
        "expires_in": 48
    },
    "method": "get",
    "params": [ ],
    "url": "https://example.com/api/auth"
}
```

Revoking a refresh token

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

DELETE /auth/ (**string:** *refresh_token*)

Revoke a *refresh token*

Request Headers

- Authorization – the *access token* as Bearer token

Parameters

- **refresh_token** (*string*) – the *refresh token* to revoke

Status Codes

- 204 No Content – the refresh token was deleted
- 400 Bad Request – the request is malformed
- 401 Unauthorized – the *access token* is invalid or
- 404 Not Found – the *refresh token* was already revoked or not exists

Objects `/objects`

Get an object

GET /objects/ (*object_id*)
Get an object detail.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search
- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Note: Note that the response data fields can change depending of BEdita object type exposed and configuration.

Example request:

```
GET /objects/15 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "api": "objects",
    "data": {
        "object": {
            "id": 15,
            "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/15/relation/attach"
                },
                "seealso": {
                    "count": 2,
                    "url": "https://example.com/api/objects/15/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/15"
}
```


Note: 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 objects) and `url` fields. The `url` is the complete API request url to get the object related, for example https://example.com/api/objects/15/relations/attach **Embedding related objects**

Requests with `embed[relations]` query string will add `objects` key to `data.object.relations.<relation_name>`, for example

```
GET /objects/15?embed[relations]=attach|3,seealso|2 HTTP/1.1
```

will have as `relations` key


```
{
    "relations": {
        "attach": {
            "count": 8,
            "url": "https://example.com/api/objects/15/relation/attach",
            "objects": [
```

```
                {
                    "id": 13,
                    "title": "attach one"
                },
                {
                    "id": 21,
                    "title": "attach two"
                },
                {
                    "id": 22,
                    "title": "attach three"
                }
            ]
        },
        "seealso": {
            "count": 2,
            "url": "https://example.com/api/objects/15/relation/seealso",
            "objects": [
                {
                    "id": 30,
                    "title": "seealso one"
                },
                {
                    "id": 31,
                    "title": "seealso two"
                }
            ]
        }
    }
}
```


where the `objects` collections have been simplified but every item inside them is a complete object.

Note: If *object_id* corresponds to a **section** or a **publication** then the response will contain `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 collection of objects

The `/objects` endpoint can be used to retrieve a collection of objects.

`GET /objects`
It returns a collection of objects:

- if called with `id` query string parameter the response will contain a collection of the objects requested
- else it returns a paginated list of objects that are descendants of the related publication configured in `app/config/frontend.ini.php`.

Important: Note that when `id` query string is used, no other parameters is valid but *access token*.

The response will be an array of objects as shown below.

Request Headers

- Authorization – optional *access token*

Query Parameters

- **`filter[object_type]`** (*string*) – the object type or a comma separated list of object types requested
- **`filter[query]`** (*string*) – used for fulltext search
- **`embed[relations]`** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **`page`** (*int*) – the page requested
- **`page_size`** (*int*) – the page dimension
- **`id`** – a comma separated list of object ids. **See the important note above**. The max number of ids you can request is defined by `ApiBaseController::$paginationOptions['maxPageSize']`

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The request is not authorized to access to protected publication
- 403 Forbidden – The request is authorized but without sufficient permissions to access to protected publication

Example request:

```
GET /objects HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

For readability the fields of objects are limited to "title" but they are similar to *GET /objects/(object_id)* example

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "api": "objects",
    "data": {
        "objects": [
            {
                "id": 2,
                "title": "title one"
            },
            {
                "id": 3,
                "title": "title two"
            },
            {
                "id": 4,
                "title": "title three"
            },
            {
                "id": 5,
                "title": "title four"
            },
            {
                "id": 6,
                "title": "title five"
            }
        ]
    },
    "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 object's children

GET /objects/ (*object_id*) **/children**

Return the paginated children of object *object_id*. The object has to be a section or the publication.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search
- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get object's children of type *section*

GET /objects/ (*object_id*) **/sections**

Return the paginated children of object *object_id*. The object has to be a section or the publication. The children are just sections (*section BEdita object type*)

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[query]** (*string*) – used for fulltext search
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get object's children of type *contents*

GET /objects/ (*object_id*) **/contents**

Return the paginated children of object *object_id*. The object has to be a section or the publication. The children are other than sections.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int* | *string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested. "*section*" object type is not accepted
- **filter[query]** (*string*) – used for fulltext search
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get object's descendants

GET /objects/ (*object_id*) **/descendants**

Return the paginated children of object *object_id*. The object has to be a section or the publication. The children are other than sections.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search
- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get object's siblings

GET /objects/(*object_id*)**/siblings**

Return the paginated siblings of object *object_id*.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search

- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get relations count

GET /objects/ (*object_id*) **/relations**

Returns a summary of relations information about object *object_id*. It shows every relation with the **count** and the **url** to get the related objects detail.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search
- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Example request:

```
GET /objects/15/relations HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "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

GET /objects/ (*object_id*) **/relations/**

string: *relation_name* Return the paginated collection of objects related by *relation_name* to object *object_id*.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)
- **relation_name** (*string*) – the name of the relation

Query Parameters

- **filter[object_type]** (*string*) – the object type or a comma separated list of object types requested
- **filter[query]** (*string*) – used for fulltext search
- **embed[relations]** (*string*) – used for embedding related objects in `relations` key. For example `embed[relations]=attach|3,seealso|2` will embed 3 objects related by "attach" and 2 related by "seealso" to main object. If no number is specified then only one relation will be embed i.e. `embed[relations]=poster` is the same of `embed[relations]=poster|1`. See the *example*.
- **page** (*int*) – the page requested
- **page_size** (*int*) – the page dimension

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Get the relation detail between objects

GET /objects/ (*object_id*) **/relations/**
string: *relation_name***/int:** *related_id* Returns the relation detail between object *object_id* and *related_id*.

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)
- **relation_name** (*string*) – the name of the relation that ties *object_id* and *related_id*
- **related_id** (*int*) – the object id of the related object

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized

- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Example request:

```
GET /objects/15/relations/attach/23 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "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

GET /objects/ (*object_id*) **/children/**
int: *child_id* Return the position (`priority` key) of *child_id* relative to all children of parent object *object_id*

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int|string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)
- **child_id** (*int*) – the object id of the child of object *object_id*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Example request:

```
GET /objects/1/children/2 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

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

Create an object

`POST /objects`

Create a new BEdita object type.

Important: To write an object it has to be *configured to be writable*

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

The request body has to be a JSON as

```
{
    "data": {}
}
```

where inside `"data"` are placed all fields to save. User has to be *authenticated* and `"data": {}` 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).

Required keys in JSON are shown below.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Object

- **data.object_type** (*string*) – (**required**) the object type to create
- **data.parents** (*array*) – (**required** if `data.relations` with conditions specified below missing) a list of parents. Parents must be accessible (with right permission for user authorized) on publication tree
- **data.relations** (*object*) – (**required** if `data.parents` with conditions specified above missing) an object of relations where the keys are the relations' names. Every relation contains an array of objects of `related_id` and optionally of relation params

```
{
    "name1": [
        {
            "related_id": 1
        },
        {
            "related_id": 2,
            "params": {
                "name_param_one": "value param one",
                "name_param_two": "value param two"
            }
        }
    ],
    "name2": [
        {
            "related_id": 3
        }
    ]
}
```

- **data.custom_properties** (*object*) – (**optional**) a list of custom properties to save. Set a custom property to `null` to delete it. For custom properties that supports multi options an array of values can be passed. Custom properties types are checked before save, so if type is *number* its value must be numeric, if it's *date* its value must be a compatibile ISO 8601 format.

```
{
    "custom_properties": {
        "custom_name_text": "my text here",
        "custom_name_number": 12,
        "custom_name_date": "2015-12-15T09:29:00+02:00",
        "custom_name_multiple": ["one", "two", "three"],
        "custom_name_to_remove": null
    }
}
```

Response Headers

- Content-Type – application/json
- Location – The url to the resource just created `https://example.com/objects/(object_id)`

Status Codes

- 201 Created – Success, the object was created. Return the object detail as in *GET /objects/(object_id)*
- 400 Bad Request – Required parameters are missing or the request is malformed

- 401 Unauthorized – Request is not authorized

Example request:

```
POST /objects HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

{
    "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"
            }
        ]
    }
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://example.com/objects/45

{
  "api": "objects",
  "data": {
      "id": 45,
      "title": "My title",
      "object_type": "event",
      "description": "bla bla bla"
  },
  "method": "post",
  "params": [],
  "url": "https://example.com/api/objects"
}
```

The response payload contains the created object detail. *In the example above only some fileds are shown.*

dates must be in ISO 8601 format.

Update an object

POST /objects

Update an existent object.

Important: To write an object it has to be *configured to be writable*

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

POST request can be also used to **update an existent object**. In that case the object `id` has to be passed in `"data"` (as *creating object*) and `object_type` can be omitted.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Object

- **`data.id`** (*string*) – (**required**) the id of the object to update

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success, the object was updated. Return the object detail as in *GET /objects/(object_id)*
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized

Create or update relations between objects

POST /objects/ (*object_id*) **/relations/**

string: *relation_name* Create relations *relation_name* between *object_id* and other objects. If the relation between objects already exists then it will be updated.

Request data must be an array of relation data or only a relation data if you need to save only one relation.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Array of Objects

- **related_id** (*string*) – (**required**) the related object id
- **params** (*string*) – (**optional**) it depends from relation type
- **priority** (*string*) – (**optional**) is the position of the relation. Relation with lower priority are shown before.

Response Headers

- Content-Type – application/json
- Location – If at least a new relation was created (201 Created). The url to *collection of related objects*

Status Codes

- 200 OK – Success but no new relation was created.
- 201 Created – Success and at least a new relation was created. Return the object detail as in *GET /objects/(object_id)*
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized

Example request to create/update only one relation:

```
POST /objects/3/relations/attach HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

Example request to create/update a bunch of relations:

```
POST /objects/3/relations/attach HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

{
    "data": [
        {
            "related_id": 15,
```

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

Example response:

```
HTTP/1.1 201 Created
Host: example.com
Location: https://example.com/objects/3/relations/attach
Accept: application/json, text/javascript
Content-Type: application/json
```

The response body will be the same as *GET /objects/(object_id)/relations/(string:relation_name)*

Replace relation data between objects

PUT /objects/(*object_id*)**/relations/**
string: *relation_name***/int:** *related_id* Replace the relation *relation_name* data between *object_id* and *related_id*.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Object

- **related_id** (*string*) – (**required**) the related object id
- **params** (*string*) – (**optional**) it depends from relation type
- **priority** (*string*) – (**optional**) is the position of the relation. Relation with lower priority are shown before.

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized

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

Example request:

```
PUT /objects/1/relations/attach/2 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json
```

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

Example response:

```
HTTP/1.1 200 Success
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json
```

The response body will be the same as *GET /objects/(object_id)/relations/(string:relation_name)/(int:related_id)*

Add or edit children

POST /objects/ (*object_id*) **/children**

Add or edit children to area/section object type identified by *object_id*

Request data must be an array of child data or only a child data if you need to save only one child.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Array of Objects

- **child_id** (*string*) – (**required**) the child object id
- **priority** (*string*) – (**optional**) is the position of the child on the tree. Relation with lower priority are shown before.

Response Headers

- Content-Type – application/json
- Location – If at least a new child was created (201 Created) it contains the url to *collection of children objects*.

Status Codes

- 200 OK – Success but all objects already were children of *object_id*. The children position (`priority`) could be changed. Response body is the children detail *GET /objects/(object_id)/children*
- 201 Created – Success and at least a new child was added to parent *object_id*. Response body is the children detail *GET /objects/(object_id)/children*.
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized

Example request to add/edit many children:

```
POST /objects/3/children HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

Example request to add/edit one child:

```
POST /objects/3/children HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

The response body will be the same as *GET /objects/(object_id)/children*

Update child position

PUT /objects/(*object_id*)**/children/**
int: *child_id* Change the child position inside the children of *object_id*.

Request Headers

- Authorization – (**required**) *access token*

Request JSON Object

- **priority** (*string*) – (**required**) the position of child object id

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success. Children position (`priority`) updated.
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized

Example request:

```
POST /objects/1/children/2 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

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

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "api": "objects",
  "data": {
    "priority": 5
  },
  "method": "get",
  "params": [],
  "url": "https://example.com/api/objects/1/children/2"
}
```

Delete an object

DELETE /objects/(*object_id*)

Delete the object *object_id*

Request Headers

- Authorization – (**required**) *access token*

Response Headers

- Content-Type – application/json

Status Codes

- 204 No Content – The object was deleted.
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized
- 404 Not Found – The object to delete not exists or it has already been removed

Example request:

```
DELETE /objects/15 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 204 No Content
```

Delete a relation between objects

DELETE /objects/ (*object_id*) **/relations/**
string: *relation_name***/int:** *related_id* Delete the relation *relation_name* between *object_id* and *related_id*

Request Headers

- Authorization – (**required**) *access token*

Response Headers

- Content-Type – application/json

Status Codes

- 204 No Content – The relation *relation_name* between *object_id* and *related_id* was deleted.
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized
- 404 Not Found – The relation *relation_name* between *object_id* and *related_id* not exists or it has already been removed

Example request:

```
DELETE /objects/10/relations/seealso/36 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 204 No Content
```

Remove a child from a parent

DELETE /objects/ (*object_id*) **/children/**
int: *child_id* Remove *child_id* from *object_id* children

Request Headers

- Authorization – (**required**) *access token*

Response Headers

- Content-Type – application/json

Status Codes

- 204 No Content – *child_id* was removed from *object_id* children.
- 400 Bad Request – Required parameters are missing or the request is malformed
- 401 Unauthorized – Request is not authorized
- 404 Not Found – *child_id* not exists or it has already been removed from *object_id* children.

Example request:

```
DELETE /objects/1/children/3 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 204 No Content
```

Files upload `/files`

It is used to upload files that must be linked to an object, generally multimedia objects as *image*, *video*, *audio*,...

The file upload flow is described in *Uploading files* section.

Warning: At the moment uploaded files are only linkable to new BEdita objects that support upload. You can't change file associated to an existing object.

Upload a file

POST /files/ (**string:** *object_type*) **/**

string: *file_name* Upload a file related to an `object_type` and named `file_name` and obtain an *upload token*. The `object_type` needs to validate some property of the file uploaded as the mime type. For example the *image* object type will accept `image/jpeg` but will reject `video/mp4`.

`file_name` should be the original file name **url encoded**.

Important: To write an object it has to be *configured to be writable*

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

User has to be *authenticated* and the request body must contain the binary file data.

Request Headers

- Authorization – (**required**) *access token*
- Content-Type – (**required**) the content type of the file
- Content-Length – (**required**) the size of the file

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success, the file was uploaded. The response body will contain the *upload token*
- 400 Bad Request – Required parameters are missing or the request is malformed or some validation fails.

- 401 Unauthorized – Request is not authorized
- 403 Forbidden – The quota available or the max number of files allowed is exceeded. The `UPLOAD_QUOTA_EXCEEDED` and `UPLOAD_FILES_LIMIT_EXCEEDED` *Error codes* helps to distinguish which quota exceeds the limit.
- 409 Conflict – The file is already present

Example request:

```
POST /files/image/foo.jpg HTTP/1.1
Host: example.com
Accept: application/json
Content-Type: image/jpeg
Content-Length: 284

raw image content
```

Example response:

```
HTTP/1.1 200 Success
Content-Type: application/json

{
  "api": "files",
  "data": {
      "upload_token": "80fdd2590b609d51873fe187d65f026e39748179"
  },
  "method": "post",
  "params": [],
  "url": "https://example.com/api/files/image/foo.jpg"
}
```

Once obtained the *upload token* it must be used to finalize the upload action creating a new object linked to the file uploaded. You can do it using *`POST /objects`* and passing the *upload token* in the request data payload.

Example

```
POST /objects HTTP/1.1
Host: example.com
Accept: application/json
Content-Type: application/json

{
    "data": {
        "title": "Uploaded via API!",
        "object_type": "image",
        "upload_token": "80fdd2590b609d51873fe187d65f026e39748179",
        "parents": [1]
    }
}
```

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "api": "objects",
  "data": {
      "id": 57,
```

```
      "title": "Uploaded via API!",
      "object_type": "image",
      "name": "foo.jpg",
      "original_name": "foo.jpg",
      "mime_type": "image/jpeg",
      "uri": "https://assets.example.com/cd/df/foo.jpg",
      "file_size": 284,
      "width": 200,
      "height": 100
  },
  "method": "post",
  "params": [],
  "url": "https://example.com/api/objects"
}
```

Note: In this example the image object created is located on publication tree. See *POST /objects* to know the required paramters creating an object.

Posters `/posters`

Warning: This endpoint is still in development and could be completely changed or removed in next versions.

Use it with caution.

Get the image representation of object *object_id* as thumbnail url

GET /posters/ (*object_id*)

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

1. if object *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 then it returns a thumbnail of the object
3. else if the object has an 'attach' relation with an image object then it returns a thumbnail of that image

Request Headers

- Authorization – optional *access token*

Parameters

- **object_id** (*int | string*) – identify a BEdita object. It can be the object id or the object unique name (nickname)

Query Parameters

- **width** (*int*) – the thumbnail width
- **height** (*int*) – the thumbnail height

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 401 Unauthorized – The object *object_id* is protected and the request is not authorized
- 403 Forbidden – The request is authorized but without sufficient permission to access object *object_id*
- 404 Not Found – Object *object_id* not found

Example request:

```
GET /posters/5 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json
```

Example response:

```
HTTP/1.1 200 Success
Host: example.com
Accept: application/json, text/javascript
Content-Type: application/json

{
 "api": "posters",
 "data": {
   "id": 5,
   "uri": "https://media.server/path/to/thumb/thumbnail.jpg"
 },
 "method": "get",
 "params": [],
 "url": "https://example.com/api/posters/5"
}
```

Get a collection of image representations

The `/posters` endpoint can be used also to retrieve a collection of image representations.

`GET /posters`

If called with `id` query string parameter the response will contain a collection of the requested *posters*

The response will be an array of posters as shown below.

Request Headers

- Authorization – optional *access token*

Query Parameters

- **`id`** – a comma separated list of object ids. The max number of ids you can request is defined by `ApiBaseController::$paginationOptions['maxPageSize']`
- **`width`** (*int*) – the thumbnail width
- **`height`** (*int*) – the thumbnail height

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 400 Bad Request – Malformed request
- 401 Unauthorized – The request is not authorized to access to protected publication
- 403 Forbidden – The request is authorized but without sufficient permissions to access to protected publication

Example request:

```
GET /posters?id=1,2,3,4,5 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "api": "posters",
   "data": [
        {
          "id": 1,
          "uri": "https://media.server/path/to/thumb/thumbnail-1.jpg"
        },
        {
          "id": 2,
          "uri": "https://media.server/path/to/thumb/thumbnail-2.jpg"
        },
        {
          "id": 3,
          "uri": "https://media.server/path/to/thumb/thumbnail-3.jpg"
        },
        {
          "id": 4,
          "uri": "https://media.server/path/to/thumb/thumbnail-4.jpg"
        },
        {
          "id": 5,
          "uri": "https://media.server/path/to/thumb/thumbnail-5.jpg"
        }
   ],


    "method": "get",
    "params": {
        "id": "1,2,3,4,5"
    },
    "url": "https://example.com/api/posters"
}
```

User profile `/me`

> **Warning:** This endpoint is still in development and could be completely changed or removed in next versions.
>
> Use it with caution.

Obtain information about authenticated user

`GET /me`
Return information about current authenticated user

Request Headers

- Authorization – (**required**) *access token*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Success
- 401 Unauthorized – The request is not authorized
- 404 Not Found –

See also:

To an index of all API requests see HTTP Routing Table

CHAPTER 10

Glossary

Access token A string granted by the authorization server used to identify the issuer of a request. The access token has to be sent to the resource server every time that the client want to access to protected resources.

BEdita REST API uses *JSON Web Token* as access token. It can be sent as `Authorization` HTTP header (preferred) using a `Bearer` scheme

```
Authorization: Bearer <token>
```

or as query string `/endpoint?access_token=<token>`

JSON Web Token

JWT JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

A JWT is composed by three parts:

- an **header** containing informations about the token type and algorithm used. It is Base64URL encoded.
- a **payload** containing informations in the form of claims (informations we want to transmit). It is Base64URL encoded.
- a **signature** used to verify the authenticity of the JWT using an valid algorithm defined by JSON Web Signature (JWS) specification (for example a shared secret HMAC).

More info here.

Refresh token An opaque token issued by the authorization server. It is useful to renew an expired *access token* without send the user credentials again. This token doesn't expire but can be revoked by *`DELETE /auth/(string:refresh_token)`*

Upload token An opaque token issued from API server that refers to a file uploaded from a client with an authenticated user and for a specific object type. It is used to link an uploaded file to a BEdita object as shown in *Uploading files* section.

The token has a an expiration date. If used after that expiration date the request will be invalid.

CHAPTER 11

Indices and tables

- genindex
- HTTP Routing Table

HTTP Routing Table

/auth

/files

/me

/objects

/posters

Index