BEdita Documentation

Release 4.0.0-alpha

Channelweb Srl, Chialab Srl

Sep 06, 2017

Contents

CHAPTER 1

Overview

BEdita is a tool to handle the data of your mobile, IoT, web and desktop applications.

It's a full fledged content management solution with:

- an HTTP REST driven server application - the **Backend** - with a complete API to model, create, modify and retrieve data
- a default progressive web application - the **Manager** - to control & manage your data

BEdita is built with CakePHP 3 and uses relational DBMS like MySQL, Postgres or SQLite in conjunction with (optional) NoSQL systems like Redis, Elastic Search and InfluxDb to boost performance and scale up to Big Data scenarios.

JSON API is the primary, although not unique, exchange data format.

Backend

The **Backend** is the core of BEdita: an HTTP REST service that exposes a set of API endpoints.

It's not a standalone application service or daemon like a DBMS, but it's an application deployed in a Web server like Apache or Nginx and accessible via standard HTTP ports.

So at first it may look like a *standard* Web application, but it's not.

It doesn't offer an HTML interface, it doesn't respond in HTML[1] but only in JSON[2]: the only way to communicate with this service is through its *JSON API* REST endpoints. See *API reference* for an overview.

To install a backend instance see *Setup*

[1] it's possible however, only for testing/debugging purposes, to activate HTML responses to use the **Backend** in a browser, see *Accept*
[2] other formats like YML (or XML) could be added in future releases.

Manager

The **Manager** is a progressive javascript web application to manage all of your data using the backend REST API.

You may see this application as the primary GUI of BEdita or as a *control panel* or *admin console*.

To operate with this application you need to login first.

The first user created during setup has the permanent *role* of *administrator* and using its credentials you may:

- create new users and assign roles to them
- create new roles
- create and modify objects - contents or tree structures that may be used in your applications
- create new object types from your application domain or extend existing types
- create relations between objects
- give access permissions to endpoints and objects to some roles
- set configuration settings and view system information and relevant events

Through its dynamic permission system keep in mind that users having different roles may perform different actions:

- some users may not be able to login in the **Manager** - e.g. standard users of your applications
- some users may not have access to some endpoints - like modeling new types, or view some object types
- some users may have a read-only access to some endpoints - they may only see certain types but not create or modify anything

Notes

CHAPTER 2

Setup

Below you will find instructions to setup BEdita4 on a typical server or desktop.

For a quick setup via Docker see *below*.

Prerequisites

- PHP 7.x (recommended) or PHP >= 5.6, with extensions *mbstring*, *intl*
- MySQL 5.7 (recommended) or MySQL 5.6, Postgres 9.5/9.6 or SQLite 3
- Composer

A web server like Apache or Nginx also recommended in general, but not strictly necessary on a development enviroment.

OS: there are no virtually constraints or limitations, but a recent Linux distribution is recommended for a server setup and Linux is the only officially supported server platform.

For a development/test setup you may choose any modern desktop/server OS. BEdita dev team uses mainly Linux and MacOSX so some strange things may happen using other systems :)

But let us know if you have any OS specific troubles, we will be happy to help.

Install

There are three basic steps to install BEdita 4

1. Create project via composer

```
$ composer create-project -s dev bedita/bedita
```

If you are using a **.zip** or **.tar.gz** release file you just need to unpack it and then run `composer install`.

2. Create an empty database

Create an empty MySQL or Posgtres database and keep track of main connection parameters like `host`, `username`, `password` and `database`. Do nothing for SQLite since a new local file will be created.

3. Run shell script to initialize the database and create first admin user

Open a shell prompt on root installation folder and write

```
$ bin/cake bedita setup
```

An interactive shell script will guide you through missing installation steps:

- database setup & schema check
- filesystem check
- admin user creation

You may see the first admin user created like a root user on Linux or MySQL: it has **administrator** *role* privileges and cannot be removed.

To setup a database connection you may also edit the main configuration file, see *Manual Setup*.

Web Server

On a development setup it's possible to use PHP builtin webserver, have a look at CakePHP Development Server Using this simple command

```
$ bin/cake server
```

This will serve the backend application at http://localhost:8765/.

This setup should be used **ONLY** in development and **NEVER** as a production setup.

For test and production setups is always advisable to use a **real** web server like Apache2 or Nginx.

Apache2

Below some instructions for **Apache 2.4**, but also other versions of Apache, like 2.2, will work.

Please make sure that **mod_rewrite** is enabled. On Ubuntu 14.04 or 16.04 or on Debian 8 you may verify it like this

```
$ more /etc/apache2/mods-enabled/rewrite.load
LoadModule rewrite_module /usr/lib/apache2/modules/mod_rewrite.so
```

On other systems with different Apache configurations this check should be similar.

A simple minimal working virtualhost configuration can look like this:

```
<VirtualHost *:80>
    ServerName api.example.com

    DocumentRoot /path/to/bedita/webroot
    <Directory /path/to/bedita/webroot>
        Options FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

Where:

- `/path/to/bedita` refers simply to the **Backend** path on filesystem
- `DocumentRoot` should point to the `webroot` folder
- `AllowOverride All` is needed to enable `.htaccess` files
- `Require all granted` allows access from anywhere, you may decide to set some restrictions based on hosts/IP

To enable CORS on virtualhost configuration you may add these lines, provided that **mod_headers** is enabled

```
Header set Access-Control-Allow-Origin "*"
Header set Access-Control-Allow-Headers "content-type, origin, x-requested-with,␣
↪authorization"
Header set Access-Control-Allow-Methods "PUT, GET, POST, PATCH, DELETE, OPTIONS"
Header set Access-Control-Expose-Headers "Location"
```

In this example:

- all origins and HTTP methods are allowed, you may want to add restrictions
- only headers used by BEdita4 are allowed
- "Location" header is exposed in response, this is useful to get URL of a newly created resource

Alternatively you can setup *CORS* configuration directly in BEdita, see *CORS*

Nginx

[TBD]

Manual Setup

To setup database connection manually or review the current connection you may edit the main default configuration file located in `config/app.php` where datasources are defined.

Look for `Datasources` array definition then modify `host`, `username`, `password` and `database` fields.

```
'Datasources' => [
    'default' => [
        'className' => 'Cake\Database\Connection',
        'driver' => 'Cake\Database\Driver\Mysql',
        'host' => 'localhost',
        //'port' => 'non_standard_port_number',
        'username' => '......',
        'password' => '......',
        'database' => '......',
        .....
    ],
]
```

Other noteworthy fields:

- `port` - populate only in case of non standard ports
- `driver` - change to `'Cake\Database\Driver\Postgres'` or `'Cake\Database\Driver\Sqlite'` accordingly

- for SQlite you need to set only an absolute local file path in `database`

Docker

You need a working Docker setup in order to pull, build or run images.

Pull official image

You can get the latest offical image build from Docker Hub like this.

```
$ docker pull bedita/bedita:latest
```

You may also use `:4-cactus` tag instead of `:latest`, they are currently synonyms. Release tags will be available soon.

Build image locally

If you want to build an image from local sources you can do it like this from BEdita root folder:

```
$ docker build -t bedita4-local .
```

You may of course choose whatever name you like for the generated image instead of `bedita4-local`.

Run

Run a Docker image setting an initial API KEY and admin username and password like this:

```
$ docker run -p 8090:80 --env BEDITA_API_KEY=1029384756 \
    --env BEDITA_ADMIN_USR=admin --env BEDITA_ADMIN_PWD=admin \
    bedita/bedita:latest
```

This will launch a BEdita4 instance using `SQLite` as its storage backend. It should become available at http://localhost:8090/home almost instantly.

Replace `bedita/bedita:latest` with `bedita4-local` (or other chosen name) to launch a local built image.

Using PostgreSQL or MySQL

Other database backends can be used with BEdita by launching the database server in a separate Docker container. You may simply pull `mysql:5.7` or `postgres:latest` official images to achieve this.

A MySQL 5.7 server can then be launched in a container with this command:

```
docker run -d --name mysql \
    --env MYSQL_ROOT_PASSWORD=root \
    --env MYSQL_DATABASE=bedita \
    --env MYSQL_USER=bedita \
    --env MYSQL_PASSWORD=bedita \
    mysql:5.7
```

Then, a BEdita instance can be configured to use MySQL as its backend launching this command:

```
docker run -d --name=bedita \
    --env DATABASE_URL=mysql://bedita:bedita@mysql:3306/bedita \
    -p 8090:80 --link mysql:mysql \
    bedita/bedita:latest
```

Notice the `DATABASE_URL` environment variable setting.

The BEdita container will automatically wait until MySQL container becomes available, then will run connect to it, launch required schema migrations, and start the Web server. The application should become available at http://localhost:8090/home in a matter of few seconds. However, depending on the responsiveness of MySQL container, this might take longer.

Logging

Logs are written to stdout and sterr, so that they can be inspected via `docker logs`. This is considered a common practice for Docker containers, and there are tools that can collect and ingest logs written this way. However, `LOG_ERROR_URL` and `LOG_DEBUG_URL` can be overwritten at container launch via `--env` flag to send logs to a different destination. For instance, one might want to launch a Logstash container, link it to BEdita container, and send BEdita logs to Logstash.

CHAPTER 3

Configuration

Configuration parameters of a BEdita4 application are usually stored in PHP files and on a database table.

We have:

- `config/app.php` file containing App, Datasource, Cache, Log and other basic *low-level* settings common to every CakePHP application
- `config` table with BEdita4 specific parameters loaded after `config/app.php`

To know more about configuration in CakePHP please read this CakePHP book chapter

In `config` table every plugin (like API and Core BEdita4 plugins) can define and load its own parameters using its own *context* (have a look at **Accept** configuration for an example).

Using configurations keep in mind these common usage rules and best practices:

- `config` table records may generally override `config/app.php`
- `config` table may not override *low-level* config settings like `Datasources`, `Cache`, `EmailTransport`, `Log`, `Error` and `App`
- `config/app.php` configurations should follow Twelve Factor app configuration principles : use environment variables for everything that is likely to vary between deploys

Below a brief BEdita4 configurations reference in alphabetical order.

Accept

Define extra content types to accept in HTTP requests using `Accept:` header other than *JSON* (`application/json`) and *JSONAPI* (`application/vnd.api+json`) that are always accepted.

Instead *HTML* (`text/html`, `application/xhtml+xml`, `application/xhtml`, `text/xhtml`) generally not, but can be accepted through this configuration. Same rule could be applied in the future to content types like *XML* and *YAML* (currently not supported).

In `config/app.php` we have

```
'Accept' => [
    'html' => filter_var(env('ACCEPT_HTML', 'false'), FILTER_VALIDATE_BOOLEAN),
]
```

In `config` table we may override it with a record with these fields

- *context*: `'api'`
- *name*: `'Accept'`
- *content*: `{ "html" :  true }`

CORS

It's possible to setup some basic CORS configuration parameters directly in BEdita4.

Using this settings please beware of possible conflicts with similar settings made on your HTTP server.

An optional 'CORS' configuration should be like this example:

```
'CORS' => [
    'allowOrigin' => '*.example.com', // string or array , also '*'
    'allowMethods' => ['GET', 'POST'], // (optional) array, also '*'
    'allowHeaders' => ['X-CSRF-Token'] // (optional) array, also '*'
]
```

Where:

- **CORS.allowOrigin** is a single domain or an array of domains
- **CORS.allowMethods** is an array of HTTP methods
- **CORS.allowHeaders** must be an array of HTTP headers

So if you want to allow every CORS call with the most permissive setting, on development and test systems, you may set:

```
'CORS' => [
    'allowOrigin' => '*',  // allow every origin
    'allowMethods' => '*', // allow every method
    'allowHeaders' => '*'  // allow every header
]
```

Pagination

Pagination settings for every API *list* response are done via `'Pagination'` key:

```
'Pagination' => [
    'limit' => 20,
    'maxLimit' => 100,
]
```

Where:

- **limit** - int - Default number of items per page as seen in *page_size* meta response and query string. Defaults to 20.

- **maxLimit** - int - Maximum acceptable items per page on a *page_size* request. Defaults to 100. This value cannot exceed 500.

Plugins

Plugins setup for your BEdita4 instance is done through `'Plugins'` configuration key:

```
'Plugins' => [
    'DebugKit' => ['debugOnly' => true, 'bootstrap' => true],
    'MyPlugin' => ['autoload' => true, 'bootstrap' => true, 'routes' => true],
]
```

Where each key is a plugin name, and for each plugin available options are:

- **debugOnly** - boolean - (default: *false*) If true load this plugin in *'debug'* mode only.
- **bootstrap** - boolean - (default: *false*) If true load the `$plugin/config/bootstrap.php` file.
- **routes** - boolean - (default: *false*) If true load the `$plugin/config/routes.php` file.
- **ignoreMissing** - boolean - (default: *false*) If true ignore missing bootstrap/routes files.
- **autoload** - boolean - (default: *false*) Whether or not you want an autoloader registered

Security

Additional security settings regarding anonymous access and *JWT* (JSON Web Tokens) are possible, even though not mandatory.

```
'Security' => [
    //....
    'blockAnonymousApps' => true,
    'blockAnonymousUsers' => true,
    'jwt' => [
        'duration' => '+2 hours',
        'algorithm' => 'HS256',
    ],
],
```

Where:

- **Security.blockAnonymousUsers** when true on each request user must be identified, anonymous requests will receive a `401 Unauthorized` response - when false anonymous read requests (GET) are possible, but identification is always required for write operations (POST, PATCH, DELETE)
- **Security.blockAnonymousApps** when true on each request application must be identified, anonymous or unknown applications will receive a `403 Forbidden` response
- **Security.jwt.duration** is the default duration of the generated JWT. Keeping this value low increases security, but increases load on server as more renew requests will be performed by clients.
- **Security.jwt.algorithm** is the encryption algorithm used to issue new tokens. Must be one of `HS256`, `HS512`, `HS384`, or `RS256`.

CHAPTER 4

Database

Schema

BEdita4 uses CakePHP Migrations plugin for schema creation and update. Schema migration files are located in `plugins/BEdita/Core/config/Migrations` and they are basically PHP files that describe database schema evolution.

With simple shell commands you can check migration status and perform schema update. See also *db_admin* shell command to perform schema check and initialization.

A MySQL schema file, provided for convenience only, is available in `plugins/BEdita/Core/config/schema/be4-schema-mysql.sql`. It's not used anywhere in BEdita but it may be used as a quick reference.

ER Diagram

A simple ER diagram is displayed here to give you a glimpse of BEdita4 core schema. Don't use it as a reference though: we will keep an updated version of this diagram, but some tables or fields may still be missing.

CHAPTER 5

Objects in BEdita

Definition

Brief definition: BEdita `objects` (see *object*) are the contents of your application, the atoms of the information asset you are building.

BEdita `objects` also represent your application data model.

Let's try to explain that with some examples: are you building an application to show a museum collection?

Your objects will be artworks, authors and maybe historical events, locations and museum rooms, but also images and videos.

All of them with some relations between them: an artwork was **createdBy** an author (or more authors), it could **represent** an historical event, it was **created** on a particular place and be **located** in a museum room or **borrowed** from another museum or gallery.

Are you building a rental car application?

Your objects will probably be cars, car companies, customers, locations of your offices and so on. Applying some relations you may have a car that has been **produced by** a car company, **rented** by a customer from an office.

You can create your custom object types from scratch but you have also a vast collection of core object types ready to use that may already fit your needs, like:

- contents like documents, events, news
- multimedia items like images, video, audio or collections of those items
- folders that contain other folders or contents to build a hierarchical and browsable structure for your data

Not everything is an `object` in BEdita, unlike what happens in Java and other platforms.

An object in BEdita is identified with these capabilities:

- you may extend it adding new properties
- you may create new object types inheriting an existing object type
- you may create semantic relations between objects

- every change on object properties is versioned
- objects share a common id space and have a unique name for every project - url friendly string identifier

Apart from objects we have four basic groups of entities

- entities like tags, categories, permissions or annotations are special entities that we assign to objects: they are not objects itself, they rather define object properties
- entities like object types, relations, properties are used to design our object model
- other entities like endpoints, configurations, applications, auth providers will be rarely seen directy by API client developers or backend users: they handle API and project behaviors
- roles are special entities used only to give users permissions on objects, endpoints or object types - they are not objects

`Users` (see *user*) instead are a special object type: you may add properties and relations to other objects, but you may not extend it with a new type.

It's a powerful yet simple model design that you may use successfully in a wide range of applications.

CHAPTER 6

Authentication & Authorization

BEdita4 is a multi-application system: it is designed to handle different client applications requests performed by multiple users on the same *project*.

For every request application and user roles should be identified: each application may have its own grants, and each user also depending on *role* assignement.

As a general rule: anonymous read operations are absolutely possibile, for write operations at least the user performing the request must be identified.

To handle these scenarios please have a look at *Security* configuration and set `blockAnonymousUsers` or `blockAnonymousApps` flags accordingly.

Authentication

User authentication

User authentication is performed using *JWT* standard. You may have a look a the official JWT introduction for a more detailed explanation of the standard.

Basically JWT enables a **token based authentication flow** using an *access token* to access resources and an opaque *refresh token* to renew the access token.

See *Authentication /auth* endpoint documentation on how you can retrieve the JWT tokens with a standard username and password user authentication. Once the token is retrieved it can be used on every request in the `Authorization` HTTP header this way:

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

Application identification

Client applications instead are identified using a simple **API KEY** mechanism. Through *Administration /admin* endpoint it's possible to set and retrieve an API KEY for each application registered for a project.

On each request you may set the application's API KEY using the `X-Api-Key` HTTP header

```
X-Api-Key: <api-key>
```

We prefer to talk about application *identification* instead of *authentication* because it's a weak authentication system: web or mobile client applications may expose this api key, or at least secrecy is not guaranteed. This is somehow similar to `implicit grant` in OAuth2.

The only real authentication is *user authentication*: you must rely on that for a **secure** authentication.

Authorization

Once user and application performing a request are identified, also as anonymous, the requested operation may or may not be allowed depending on the outcome of the following checks:

- **anonymous** user and application settings: depending on *Security* configuration anonymous requests can be blocked with 401 or 403 responses
- **endpoint** permissions: each endpoint may allow or deny a request from a user using a client application
- **object** permissions: at the most granular level a single *object* may or may not be accessible or modifiable to the requesting user and application

Endpoint permissions

Operation grants on endpoints can be controlled through a set of rules involving roles, applications and permission types.

Possible values for endpoint, role and application in these rules are:

- **endpoint** endpoint id or `NULL` for every endpoint
- **role** role id or `NULL` for every role or anonymous user
- **application** application id or `NULL` for every application

Instead `permission type` may have four different values for read operations (GET) or write operations (POST, PATCH, DELETE):

- **false** `(0b00)`: no permissions granted
- **true** `(0b11)`: full permissions granted
- **mine** `(0b01)`: permissions granted only on **my** resources, i.e. resources belonging to the autenticated user
- **block** `(0b10)`: no permissions granted, and override all other permissions

To better understand how these rules work an example is given below:

endpoint	role	application	permission
documents	NULL	ios-app	read: mine - write: mine
documents	manager	backend	read: true - write: true
payments	app	NULL	read: block - write: block
events	reader	web-app	read: true - write: false

- for every role (NULL) on `/documents` endpoint through `ios-app` application only resources belonging to authenticated user may be read and written
- users with `manager` role accessing `/documents` with `backend` application are able to read and modify write every resource

- users with `app` role accessing `/payments` with every application are not allowed to read or write anything
- users with `reader` role on `/events` usgin `web-app` application have a read-only access

See *Administration /admin* endpoint reference on how you may set these permission rules.

Keep in mind that altough powerful these rules must be applied very carefully like firewall network rules: it is quite easy to cause unintended side effects like blocking every operation or allow dangerous ones.

Object permissions

This feature is not yet implemented.

OAuth2

And what about **OAuth2**? At this moment BEdita4 is not (yet) an OAuth2 compliant server solution even if many OAuth2 concepts like *access token* and *refresh token* are already implemented.

Nonetheless integration with **OAuth2** Authorization services is already in development stage. A server compliant implementation will follow.

CHAPTER 7

Shell commands

A number of administration tasks can be performed via shell.

All shells and commands listed below can be invoked via a binary that's located in the `bin/` directory. The basic syntax is:

```
$ bin/cake SHELL [COMMAND] [argument1 argument2 ...] [--option1 --option2=val ...]
```

Warning: Windows users must use `bin\cake` instead (note the backslash).

For instance, if you want to run the *setup* command in the *bedita* shell you can run:

```
$ bin/cake bedita setup
```

If you want to run the *init* command in the *db_admin* shell with a couple of additional options you can run:

```
$ bin/cake db_admin init --no-force --seed
```

Note: Bash autocomplete might come in handy

bedita

The `bedita` shell is the first place you'll ever want to search in when you're looking for a tool to perform an administrative task.

setup

The `setup` subcommand will lead you through the several steps required to get a new BEdita instance to work.

You'll be asked for database connection credentials, as well as user credentials for the first administrator user to be added to the system.

Additional steps, like checking for correct permissions on some folders, creating database tables, or checking if current database schema is up-to-date with the latest changes, are performed.

It is safe to run this command again once BEdita has been initialized.

-y, --yes
Run in non-interactive mode.

--config-file <path>
Path to configuration file (default: `config/app.php`).

Example

```
$ bin/cake bedita setup
```

db_admin

The `db_admin` shell is specialized in database administration tasks for developers. As a user of BEdita, you'll hardly need any of the commands that this shell provides.

init

This subcommand will initialize the database. The database connection **must** be already configured in order for this command to work. If you're looking for a command-line wizard to guide you through all the steps required to install BEdita, please head to the *setup command*.

If any table is present in the database, you'll be asked if you want your database to be weeped, or if you wish to abort the operation. When the schema has been created, you'll also be asked if you want additional set of data to be loaded into the database. A minimum set of data is loaded regardless of your choice — this is required in order for BEdita to work.

This command can be run in unattended (non-interactive) mode by appending command-line flags `--[no-]force` and `--[no-]seed`.

-f, --force
Force removal of all tables in case target database is not empty.

--no-force
Abort if target database is not empty.

-s, --seed
Seed database with additional set of data.

--no-seed
Don't load any additional set of data. A minimum set of data required for BEdita to work is loaded anyway.

-c <connection>**, --connection** <connection>
Database connection to be used (default: `default`).

Example

```
$ bin/cake db_admin init --no-force --seed
```

check_schema

This subcommand will perform checks on the current schema. This command is mostly useful when developing features that require making changes to the schema of BEdita's database.

You'll be notified of:

- migration history not in sync (schema not migrated to the latest available version)
- new changes (added or removed tables; changes to columns, indexes or constraints)
- naming that offends SQL conventions

This command exits with a non-zero exit code whenever current schema is not completely up-to-date and it follows SQL conventions, making it possible to employ this command in other automated tasks.

-c `<connection>`, **--connection** `<connection>`
Database connection to be used (default: `default`).

Example

```
$ bin/cake db_admin check_schema --verbose
```

jobs

The `jobs` shell is used to run asynchonous pending jobs stored in *async_jobs* table.

pending

The `pending` subcommand will launch all *pending* jobs waiting to be run. A `--limit` option may be used to set a maximum number of jobs to run.

-l `<limit>`, **--limit** `<limit>`
Specify maximum number of jobs to run.

Example

```
$ bin/cake jobs pending -l 10
```

run

The `run` subcommand will launch a single job from its UUID.

Example

```
$ bin/cake jobs run 0660d795-d1bf-4ca0-9a05-2ee47943a658
```

spec

The `spec` shell can be used to automatically generate piece of documentation for available API endpoints.

generate

The `generate` subcommand will generate Swagger documentation in YAML format for currently available API endpoints.

`-o` `<output>`**, `--output`** `<output>`
Specify an output file (default: `plugins/BEdita/API/spec/be4.yaml`).

Example

```
$ bin/cake spec generate
```

CHAPTER 8

API reference

Introduction

BEdita 4 exposes a set of default endpoints to model and handle your data and to perform many other tyipical backend API operations. Throughout this documentation a description of each of these **core** endpoints is given.

All API examples are conventionally described using only **HTTP** requests and responses i.e. methods, url, headers and bodies. There are intentionally no API client language specific examples, so a basic knowledge of REST API concepts and HTTP is necessary.

Headers

Since the primary exchange format is *JSON API* its content negotiation rules are applied with some extensions.

Every request must contain an `Accept` header with one of the following values:

- `application/vnd.api+json` *JSON API* default content type
- `application/json` is also fine; considered a synonym for *JSON API*, is treated the same way

Under special circumstances, and only for GET requests, also `text/html`, `application/xhtml+xml`, `application/xhtml` and `text/xhtml` may be valid, provided that HTML accept is configured (see *Accept*): response will be an HTML version of the JSON body. Main use is enable data navigation using a browser only in test and development enviroments.

Request containing input data like POST, PATCH and in some cases DELETE must also provide a `Content-Type` header where accepted values are:

- `application/vnd.api+json` *JSON API* default content type
- `application/json` is considered a synonym for *JSON API* just like `Accept`

Note: some special requests like `POST /auth` or `POST /signup` will require `application/json` since not *JSON API* compliant, see ref:*auth-login* or ref:*signup-request* for some examples.

Other two important headers are:

- `Authorization` containing JWT token in the form `Bearer {token}` to identify the user performing the request
- `X-Api-Key` containing a token to identify the application performing the request

Typical requests will then be like:

```
POST /users HTTP/1.1
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9......
```

```
GET /users/123 HTTP/1.1
Accept: application/json
X-Api-Key: WF0IjoxNDk1MTgzMjgzLCJuYmYiOjE0OTUxODMyODMsInN1
```

Dynamic endpoints

Every *object* type in BEdita defines a new endpoint dynamically, so there will be more `dynamic` endpoints in a typical project setup than the ones described here.

Some core object types like `documents` or `profiles` already provide their endpoints namely `/documents` and `/profiles` where through GET/POST/PATCH/DELETE methods you will be able to manage their data in the same way you can do in *Users /users*.

The only difference will be data structure and relations between objects that will of course vary from type to type hence from endpoint to endpoint.

Same will happen for new object types created dynamically in *Object types /object_types*: a new dedicated endpoint is created, so a new *cats* object type will automatically have its own `/cats` endpoint.

Request and Response

Every **BEdita 4** API call provides a standard request and response format layout: most of the endpoints are operations involving resources, think of classical CRUD operations for simplicity, and will have the same layout while some other operations like authentication are not and are slightly different.

Requests and responses will follow in general the *JSON API* specification as default format. There are few exceptions though: in some situations a very simple JSON object format is used as request body. See below for some examples.

Methods

HTTP methods used in API requests, following common REST best practices, are:

- **GET** to read resources or other meta informations
- **POST** to create new resources or for operations requiring input data like authentication
- **PATCH** to modify existing resources
- **DELETE** to delete existing resources

Request

DELETE and **GET** requests do not contain in general a body. Have a look at *Common GET parameters* for an overview of GET query strings.

In case of **POST** and **PATCH** methods instead a *JSON API* compliant format body part has to be sent by the client.

A minimal body request will have this layout:

```
{
    "data": {
        "type": "documents",
        "attributes": {
            "title": "This is a title",
            "description": "Brief description or abstract",
            "body": "Document body with long text..."
        }
    }
}
```

Very few operations like authentication require a flat key-value JSON object format like:

```
{
    "username": "my-username",
    "password": "my-password"
}
```

Response

Everytime a response contains a body it will be in *JSON API* compliant format.

A simple example of a **GET /documents** response will help us highlight some important concepts:

```
{
    "data": [
        {
            "id": "2",
            "type": "documents",
            "attributes": {
                "title": "My first document",
                "description": "A brief description",
                "body": "A long description",
                "status": "draft",
                "uname": "documents_my-first-document",
                "custom_field": 1234,
                "extra": {
                    "ke1": "value1",
                    "key2": [
                        "value2",
                        "other value"
                    ]
                },
                "lang": "en"
            },
            "meta": {
                "locked": false,
                "created": "2017-03-22T17:39:23+00:00",
                "modified": "2017-06-25T21:33:33+00:00",
```

```
                "published": null,
                "created_by": 3,
                "modified_by": 9
            },
            "links": {
                "self": "https://api.example.com/documents/2"
            },
            "relationships": {
                "linked_to": {
                    "links": {
                        "related": "https://api.example.com/documents/2/
↪linked_to",
                        "self": "https://api.example.com/documents/2/
↪relationships/linked_to"
                    }
                }
            }
        },
        {
            "id": "3",
            "type": "documents",
            "attributes": {
            }
        },
        {
        }
    ],
    "links": {
        "self": "https://api.example.com/documents",
        "home": "https://api.example.com/home",
        "first": "https://api.example.com/documents",
        "last": "https://api.example.com/documents&page=5",
        "prev": null,
        "next": "https://api.example.com/documents&page=2"
    },
    "meta": {
        "pagination": {
            "count": 92,
            "page": 1,
            "page_count": 5,
            "page_items": 20,
            "page_size": 20
        }
    }
}
```


This is the classical response in case of resources list. For a single resource response is very similar, but somehow reduced.

At the root level we have:

- `"data"` key will contain an array of resources, or a single resource when a single resource is requested like in **GET /documents/2**
- `"meta"` will contain metadata of single resources or of resource lists, mainly pagination info for lists
- `"links"` key appears in more than one place and will contain
  - `"self"` (always present) with a link pointing to the resource containing it; it can be an array of resources, a single resource or a relationship

  - `"home"` link to *Home Document /home* endpoint
  - `"first"` `"last"` `"prev"` `"next"` *Pagination* URIs to access pages in case of resources array

Giving a closer look at **data** key there are some important things to notice:

- `"id"` and `"type"` must always be present, `"type"` is the unique plural name of object types (like *documents* in this case) or of internal resources (like *roles*)
- `"attributes"` is for resource attributes, these are generally properties that a client application can modify, once permissions are granted, and contains
  - **stantard core** properties like `"title"`, `"description"`, `"lang"` or `"body"` available for all object types
  - **custom** properties that belong to the current resource or object type only - `"custom_field"` in this example
  - **special core** properties available for all object types, with some specific meaning and usage:
    * `"status"` may have only `on`, `draft` or `off` as values; it representes an editorial status, depending on client application settings objects with status `draft` and `off` may not be visible
    * `"uname"` is the object unique name, a modifiable unique identifier of every object inside a project, represented by an URL-friendly string that may be derived from the `"title"` or from other properties; upon creation a `uname` is automatically created and when a client application requests to create or modifiy this property the actual value may be changed by the system to avoid conflicts with an existing identical `uname`
    * `"extra"` is a special property where an application may put arbitrary data, use it as you wish... we don't wanna know :)
- `"meta"` is for resource metadata, or properties that are not directly changeable by a client application, like:
  - `"created"`, `"modified"` and `"published"` will show creation date, last modification date and date of publishing (when `status` changed to `on`)
  - `"created_by"` and `"modified_by"` display the id of the user that created the object and of the user that modified it for the last time
  - `"locked"` is a special property, when set to **true** an object is not deletable and its `status` and `uname` cannot be modified
- `"links"` contains a `"self"` link to representation of the object itself
- **`"relationships"` displays links to read or manipulate related resources or objects, in general we have:**
  - **core** relations between objects and resources, like the one between *user* and *role*
  - **custom** semantic relations between objects dynamically created using *Relations /relations*

Status codes

Obviously on every response a meaningful *HTTP status code <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>* is returned. The API consumer must always check it before reading the response body: especially to see if an error has occured.

Here a short list of the main status codes your client application will receive:

- `200 OK` - A successful GET or PATCH or POST that doesn't result in a creation.
- `201 Created` - Successful POST that results in a creation. A Location header pointing to the location of the new resource will be provided.

- `204 No Content` - Successful not returning a body (like a DELETE).
- `400 Bad Request` - The request is malformed.
- `401 Unauthorized` - No authentication or invalid authentication details are provided.
- `403 Forbidden` - Authentication succeeded but authenticated user doesn't have access to the resource.
- `404 Not Found` - A non-existent resource is requested.
- `405 Method Not Allowed` - An HTTP method is being requested that isn't allowed (like requesting a DELETE on *Home Document /home*).
- `406 Not Acceptable` - A content negotiation error, content generated is not acceptable according to the *Accept* headers sent in the request - see *Headers*.
- `415 Unsupported Media Type` - An unsupported or unknown content type was provided as part of the request.
- `500 Internal Server Error` - Something bad happened, hope you don't get a lot of them :)

Errors

In case of error this is the expected response body layout

```
{
    "error": {
        "status": "401",
        "title": "Unauthorized",
        "code": "expired_token",
        "detail": "JWT access token has expired",
        "meta": {
            "trace": [
                "#0 /var/www/bedita/plugins/BEdita/API/src/Auth/
↪EndpointAuthorize.php(133): BEdita\\API\\Auth\\JwtAuthenticate->
↪unauthenticated(Object(Cake\\Http\\ServerRequest),␣
↪Object(Cake\\Http\\Response))",
                "#1 /var/www/bedita/plugins/BEdita/API/src/Auth/
↪EndpointAuthorize.php(115): BEdita\\API\\Auth\\EndpointAuthorize->
↪unauthenticate()",
                "#2 ...",
                "....",
                "#27 {main}"
            ]
        }
    },
    "links": {
        "self": "https://api.example.com/documents",
        "home": "https://api.example.com/home"
    }
}
```

Main difference between a failure and a successful response is the presence of the `"error"` key providing:

- `"status"` response HTTP status code, for convenience
- `"title"` short, human-readable summary of the problem that SHOULD NOT change from occurrence to occurrence of the problem
- `"code"` API specific error code in string format, used by API consumer to better identify the problem occurred

- `"detail"` human-readable explanation specific to this occurrence of the problem. Can be localized.
- `"meta"` will contain stacktrace in `"trace"` only in **debug** mode
- `"links"` with `"self"` URL where error originated, and `"home"` URL

Common GET parameters

Many HTTP query parameters in BEdita4 API GET method calls are common and may be used to manipulate API response and get only the data you need in your application.

Common parameters described here may be used in all API endpoints that handle entities/objects directly.

They are available in all endpoints except few special ones like: `/auth`, `/home`, `/signup` and `/status`. These endpoints are designed to handle particular use cases where lists of resources or objects are not directly handled.

Pagination

- `page` - select page number to retrieve
- `page_size` - define page size; default page size and maximum possible size are defined via configuration parameter (defaults are 20 and 100) see *Pagination*

Examples:

- `/objects?page=2` - get page number 2
- `/objects?page_size=42` - impose 42 as page size in response
- `/objects?page_size=42&page=3` - get page number 3 using 42 as page size

In every GET response containing lists pagination data is available in a special section under `meta / pagination`

Example response:

Suppose we have 125 items available for our request, using 20 as `page_size`, selecting page number 7 we get

```
{
    "links": {
        ...
    },
    "data": {
        ...
    },
    "meta": {
        "pagination": {
            "count": 125,
            "page": 7,
            "page_count": 7,
            "page_items": 5,
            "page_size": 20
        }
    }
}
```

Where, as you may easily guess:

- `count` is the total number of items in response

- `page` is the current page number
- `page_count` is the total number of pages
- `page_items` is the numer of items displayed in this page (maximum is `page_size`)
- `page_size` is the page size, max number of items for each page

Field selection

- `fields` - decide which fields are displayed in a response, comma separeted list

Search

- `q` or `filter[query]` - perform a natural language search using a filter, see *Query filter* below

Sort

- `sort` - sort items using a field in ascending or descending order, descending order is obtained prepending minus – to field name

Examples:

- `/users?sort=username` return users alphabetically by username
- `/users?sort=-username` return users alphabetically from last to first by username (reverse order)
- `/roles?sort=-id` roles from last to first id

Filters

Filters are one the most powerful and versatile tools in **BEdita API** in order to retrieve only resources or objects you really need in your application.

Filter expressions

Generally every filter in a query string will have following syntax:

1. `filter[{item}]={value}` the simplest form, a filter item should be equal to some value
2. `filter[{item}][{op}]={value}` is a logical operation on filter value, where `{op}` can have following values:
   - `neq`, `ne`, `!=` or `<>` negation operator, meaning that filter item shoud not be equal to `{{value}}`
   - `lt` or `<` less than operator, meaning that filter item shoud be less than `{{value}}`
   - `lte`, `le` or `<=` less than or equal operator; filter item shoud be less than or equal to `{{value}}`
   - `gt` or `>` greater than operator; filter item shoud be greater than `{{value}}`
   - `gte`, `ge` or `>=` greater than or equal operator; filter item should be greater than or equal to `{{value}}`
3. `filter[{item}][]={value}` array sintax: add a value to an array representing possible values of a filter item

Filter expressions can be combined using `&` separator as many times as you want, limited only by the URL size.

Too many `filter` combinations may of course result in unwanted or meaningless results, use them with caution :)

Field filter

The simplest and most common filter: retrieve only resources that have a field equal to some value, or greater/less than some value.

Examples:

- `/users?filter[name]=Gustavo` get users that have *Gustavo* as first `name`
- `/objects?filter[id][gt]=100` return users with `id` greater than 100

Query filter

Simple text search may be performed with a query filter

- `/objects?filter[query]=gustavo` get objects that have *gustavo* in some of their textual fields
- `/objects?q=gustavo` convience alias for the preceeding filter query - `filter[query]=..` or `q=..` are identical

Note: currently only raw text search is performed, more sophisticated actual natural language search will be available in a future release

Type filter

Type filters are used to select some *object* types, typically used in `/objects` endpoint

- `/objects?filter[type]=events` select only objects of type *events*
- `/objects?filter[type][ne]=documents` all object types except *documents*
- `/objects?filter[type][]=locations&filter[type][]=profiles` select only *locations* and *profiles*

Geo filter

Geo filters are able to retrieve results on objects of type `location` or on types extending `locations` using geo coordinates.

- `/locations?filter[geo][center]=44.4944183,11.3464055` retrieve locations ordered by proximity to a given `center` point expressed in terms of latitude and longitude; each item will show in `meta.extra.distance` the distance in meters to the `center` point
- `/locations?filter[geo][center]=44.4944183,11.3464055&filter[geo][radius]=5000` same as the above filter, but results are limited in a radius of 5km

Note: in order to work this filter **requires** that the underlying database supports geo-spatial functions like `ST_GeomFromText`, this is true for **MySQL 5.7** or **PostGIS** for instance.

Date ranges filter

Date ranges are entities used in some objects like `events` to indicate start and end dates.

With this filter you are able to retrieve objects using conditions on date time intervals.

- `/events?filter[date_ranges][start_date][gt]=2017-08-01` events starting after `2017-08-01`
- `/events?filter[date_ranges][end_date][le]=2017-08-15` events with end date lesser than or equal to `2017-08-15`
- `/events?filter[date_ranges][start_date][gt]=2017-07-01&filter[date_ranges][end_date][l` events starting and ending between `2017-07-01` and `2017-07-30`

Home Document `/home`

`/home` endpoint returns basic information about available API resources

Inspired by **Home Document for HTTP APIs** IETF draft http://mnot.github.io/I-D/json-home/ its purpose is to show avilable endpoints, content type formats and HTTP methods

This information depends on user permissions and endpoint configuration.

Example request:

```
GET /home HTTP/1.1
Host: example.org
Accept: application/json
```

Example response:

```
HTTP/1.1 200 Success
Host: example.org
Content-Type: application/json

  {
      "links": {
          "self": "http://example.org/home",
          "home": "http://example.org/home"
      },
      "meta": {
          "resources": {
              "/users": {
                  "href": "http://example.org/users",
                  "hints": {
                  "allow": [
                      "GET"
                  ],
                  "formats": [
                      "application/json",
                      "application/vnd.api+json"
                  ]
              },
              "/roles": {
                  "href": "http://example.org/roles",
                  "hints": {
                  "allow": [
```

```
                    "GET"
                ],
                "formats": [
                    "application/json",
                    "application/vnd.api+json"
                ]
            },
        }
    }
}
```

Authentication `/auth`

`/auth` endpoint is responsible for login related actions. With it a user is able to login and to get user login data.

Different actions are supported:

- standard **login** - providing username and password and without `Authorization` header
- token **renew** - with `Authorization` header and without username and password
- **whoami** - get logged user profile data
- profile **update** - logged user profile data
- credential **change** request - request to update password
- credential **change** update - actual password update action using secret hash

Login

`POST /auth`

Perform login.

Form Parameters

- **`username`** – Username of user to be logged in. To be omitted when renewing token.
- **`password`** – Password of user to be logged in. To be omitted when renewing token.

Request Headers

- Authorization – *(Optional)* Use only in token renew, prefixed with `Bearer`.

Status Codes

- 200 OK – Login successful.
- 401 Unauthorized – Unauthorized user, or invalid renew token.

Response JSON Object

- **`meta.jwt`** – The JSON Web Token to be used to authenticate (in header Authorization).
- **`meta.renew`** – The renew token, to be used to reiterate login process when JWT expires.

Example request (login with username and password): since this is not a *JSON API* request you MUST use `Content-Type: application/json` or `Content-Type: application/x-www-form-urlencoded`, see example below.

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/json

{
    "username" : "bedita",
    "password" : "bedita"
}
```

Same request using classic `Content-Type: application/x-www-form-urlencoded`

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/x-www-form-urlencoded

username=bedita&password=bedita
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "links": {
        "self": "http://example.com/auth",
        "home": "http://example.com/home"
    },
    "meta": {
        "jwt": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJpZCI6MSwidXNlcm5hbWUiOiJiZWRpdGEiLCJibG9ja2VkIjpmYWxzZSwibGFzdF9sb2dpbiI6IjIwMTYtMDgtMDFUMT
↪rGcCEKiYjETnkaKVgG5-gJxIMXALVaZ4MeV5aKbWtQE",
        "renew": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJpc3MiOiJodHRwOlwvXC8xMC4wLjgzLjQ6ODA4MCIsImlhdCI6MTQ3MDA1ODUxNywibmJmIjoxNDcwMDU4NTE3LCJzdW
↪mU3QToPvc0uY-XQRhDA1F2hfpRjjT2ljSerKQygk2T8"
    }
}
```

Example request (token renewal):

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
↪eyJpc3MiOiJodHRwOlwvXC8xMC4wLjgzLjQ6ODA4MCIsImlhdCI6MTQ3MDA1ODUxNywibmJmIjoxNDcwMDU4NTE3LCJzdW
↪mU3QToPvc0uY-XQRhDA1F2hfpRjjT2ljSerKQygk2T8
```

Example response: (*same as above, with new JWT and renew token*)

Who Am I?

`GET /auth/user`
Get logged user profile data.

Request Headers

- Authorization – Use token prefixed with `Bearer`.

Status Codes

- 200 OK – Get operation successful.
- 401 Unauthorized – Unauthorized user, user not logged.

Response JSON Object

- **`data`** – User profile data

Example request

{token} is JWT token from previuos login and renew examples:

```
GET /auth/user HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Authorization: Bearer {token}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
        "id": "2",
        "type": "users",
        "attributes": {
            "username": "gustavo",
            "blocked": false,
            "last_login": "2016-10-06T08:17:36+00:00",
            "last_login_err": null,
            "num_login_err": 0,
            "name": "Gustavo",
            "surname": "Supporto"
        }
    },
    "links": {
        "self": "http://example.com/auth/user",
        "home": "http://example.com/home"
    },
}
```

Note: some fields in *"attributes"* are missing for brevity.

Update user profle

`PATCH /auth/user`

Update logged user profile data with some restrictions. For basic security reasons some fields are not directly changeable: *username*, *email* and *password*.

Request Headers

- Authorization – Use token prefixed with `Bearer`.

Status Codes

- 200 OK – Get operation successful.

- 401 Unauthorized – Unauthorized user, user not logged.

Response JSON Object

- **data** – User profile data

Example request

{token} is JWT token from previuos login and renew examples:

```
PATCH /auth/user HTTP/1.1
Host: example.com
Authorization: Bearer {token}
Accept: application/vnd.api+json
Content-Type: application/json

{
    "city" : "Bologna",
    "country" : "Italy"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
        "id": "2",
        "type": "users",
        "attributes": {
            "username": "gustavo",
            "name": "GUstavo",
            "surname": "Supporto"
            "city" : "Bologna",
            "country" : "Italy"
        }
    },
    "links": {
        "self": "http://example.com/auth/user",
        "home": "http://example.com/home"
    },
}
```

Note: some fields in *"attributes"* are missing for brevity.

Credentials change

Authentications credential change works in two steps:

- a credential change request action
- an actual credential change using a secret hash

Only use case currently supported is `password` change.

After a request action an email is sent to requesting user containing a URL with a secret hash to actually perform the change.

POST /auth/change
Request a credential change.

Form Parameters

- **contact** – Email of user requesting credendials change.
- **change_url** – Change URL that will be sent via email.

Status Codes

- 204 No Content – No content on operation success.
- 400 Bad Request – On malformed or missing input data.
- 404 Not Found – If no user is found.

Example request: Since this is not a *JSON API* request you MUST use `Content-Type: application/json`

```
POST /auth/change HTTP/1.1
Content-Type: application/json

{
    "contact": "{my email}",
    "change_url": "{change url}"
}
```

A `change_url` is required in order to create the URL that will be sent to the user in this form:

```
{change_url}?uuid={uuid}
```

Where `{uuid}` is a system generated hash that will expire after 24h.

In your `change_url` page you will have to read the `uuid` query parameter and proceed to actual change performing the following request.

PATCH /auth/change

Perform a credential (password) change.

Form Parameters

- **uuid** – Secret UUID sent via email in `change_url`.
- **password** – New user password.
- **login** – Optional boolean parameter, if `true` a login is also performed.

Status Codes

- 200 OK – On operation success.
- 400 Bad Request – On malformed or missing input data.
- 404 Not Found – Not found if provided UUID is incorrect or expired.

Example request: Since this is not a *JSON API* request you MUST use `Content-Type: application/json`

```
PATCH /auth/change HTTP/1.1
Content-Type: application/json

{
    "uuid": "{uuid}",
    "password": "{new password}",
    "login": true
}
```

Response will contain user data as in previous *Who Am I?* request.

If `"login"` is true a login is also performed and *JWT access token* and *refresh token* tokens are returned in `"meta"` section for immediate use. This key is optional, if missing `"login": false` is assumed.

Users `/users`

You can manage user data by using `/users` endpoint. It provides users management: creation, data retrieval, modification, removal and roles associations.

Create a user

You can create user data by using `POST /users` endpoint. Users data must be specified inside body JSON data.

Request body structure is:

```
{
    "data": {
        "type": "users",
        "attributes": {}
    }
}
```

`POST /users`

Example request (create user john doe):

```
POST /users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "type": "users",
        "attributes": {
            "username": "johndoe",
            "password": "j04nd0e",
            "uname": "johndoe"
        }
    }
}
```

Example above show all required attributes.

Expected response is `HTTP/1.1 201 OK`, with `application/vnd.api+json` body data representing the user just created.

When user already exists or data is not valid (i.e. data lacks of required fields), POST fails and response is `400 Bad Request - Invalid data`.

Successful response example follows:

```
HTTP/1.1 201 OK
Content-Type: application/vnd.api+json

{
```

```
    "data": {
        "id": 19283,
        "type": "users",
        "attributes": {
            "username": "johndoe",
            "password": "j04nd0e",
            "uname": "johndoe",
            "created_by": 1,
            "modified_by": 1,
            "created": "2016-12-13T12:08:02+00:00",
            "modified": "2016-12-13T12:08:02+00:00"
        },
        "relationships": {
            "roles": {
                "links": {
                    "related": "http://example.com/users/19283/roles",
                    "self": "http://example.com/users/19283/relationships/roles"
                }
            }
        }
    },
    "links": {
        "self": "http://example.com/users",
        "home": "http://example.com/home"
    }
}
```

`data.attributes` object contains more internal attributes.

Get user data

You can obtain user data by using `GET /users` and `GET /users/(user_id)` endpoint.

`GET /users`

It returns a collection of users:

- use `id` query string parameter to retrieve a single user by id
- use `id` query string parameter and `roles` token to retrieve user roles by user id

`GET /users/` (*user_id*)

Example request (get users):

```
GET /users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "data": [
        {
            "id": 19283,
            "type": "users",
```

```
            "attributes": {
                "username": "johndoe",
                "name": "john",
                "surname": "doe"
            }
        }
    ],
    "links": {
        "self": "http://example.com/users/19283",
        "home": "http://example.com/home",
        "first": "http://example.com/users",
        "last": "http://example.com/users",
        "prev": null,
        "next": null
    },
    "meta": {
        "pagination": {
            "count": 1,
            "page": 1,
            "page_count": 1,
            "page_items": 1,
            "page_size": 20
        }
    }
}
```

`data` is an array of objects; in this example, you see only one. `data.attributes` object contains more internal attributes.

`GET /users/` (*user_id*) **`/roles`**

You can obtain user roles by using `GET /users/(user_id)/roles` endpoint.

Example request (get user johndoe roles):

```
GET /users/19283/roles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "data": [
        {
            "id": "7",
            "type": "roles",
            "attributes": {
                "name": "dummy role",
                "description": null,
                "unchangeable": false,
                "created": "2016-12-13T11:28:32+00:00",
                "modified": "2016-12-13T11:28:32+00:00"
            },
            "links": {
                "self": "http://example.com/roles/7"
            },
```

```
            "relationships": {
            "users": {
                "links": {
                "related": "http://example.com/roles/7/users",
                "self": "http://example.com/roles/7/relationships/users"
                }
            }
            }
        }
    ],
    "links": {
        "self": "http://example.com/users/19283/roles",
        "home": "http://example.com/home",
        "first": "http://example.com/users/19283/roles",
        "last": "http://example.com/users/19283/roles",
        "prev": null,
        "next": null
    },
    "meta": {
        "pagination": {
            "count": 1,
            "page": 1,
            "page_count": 1,
            "page_items": 1,
            "page_size": 20
        }
    }
}
```

Modify a user

You can modify a user by using `PATCH /users/(user_id)` endpoint.

`PATCH /users/`(*user_id*)

Example request (modify user john doe):

In this example, purpose is modifying 'johndoe' user's name and surname from 'john doe' to 'Johnny Doe'.

```
PATCH /users/19283 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "id": 19283,
        "type": "users",
        "attributes": {
            "name" : "Johnny",
            "surname" : "Doe"
        }
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "data": {
        "id": 19283,
        "type": "users",
        "attributes": {
            "username": "johndoe",
            "name": "Johnny",
            "surname": "Doe"
        },
        "relationships": {
            "roles": {
                "links": {
                    "related": "http://example.com/users/19283/roles",
                    "self": "http://example.com/users/19283/relationships/roles"
                }
            }
        }
    },
    "links": {
        "self": "http://example.com/users/19283",
        "home": "http://example.com/home"
    }
}
```

`data.attributes` object contains more internal attributes.

Remove a user

You can delete a user by using `DEL /users/(user_id)` endpoint.

`DELETE /users/` (*user_id*)

Example request (delete user john doe):

Note: in this example user id is 19283.

```
DELETE /users/19283 HTTP/1.1
Host: example.com
```

Expected response is `204 No Content`. When user is not found, response is `404 Not Found`.

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

Add a role

You can add a role by using `POST /users/(user_id)/relationships/roles` endpoint. `(user_id)` is a placeholder for user object id. You specify role id inside JSON body passed to request.

`POST /users/` (*user_id*) **`/relationships/roles`**

Example request (add role 7 to john doe user):

In this example, purpose is adding a role (id 7) to 'johndoe' user (id 19283).

```
POST /users/19283/relationships/roles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "type": "roles",
        "id": 7
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "links": {
        "self": "http://example.com/users/19283/relationships/roles",
        "home": "http://example.com/home"
    }
}
```

Remove a role

You can remove a role by using `DELETE /users/(user_id)/relationships/roles` endpoint. `(user_id)` is a placeholder for user object id. You specify role id inside JSON body passed to request.

`DELETE /users/` (*user_id*) **`/relationships/roles`**

Example request (remove role 7 to john doe user):

In this example, purpose is removing a role (id 7) from 'johndoe' user (id 19283).

```
DELETE /users/19283/relationships/roles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "type": "roles",
        "id": 7
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "links": {
        "self": "http://example.com/users/19283/relationships/roles",
        "home": "http://example.com/home"
```

```
    }
}
```

Roles `/roles`

You can manage user roles by using `/roles` endpoint. It provides user roles management: creation, data retrieval, modification, removal and user association.

Create a role

You can create role data by using `POST /roles` endpoint. Roles data must be specified inside body JSON data.

Request body structure is:

```
{
    "data": {
        "type": "roles",
        "attributes": {}
    }
}
```

`POST /roles`

Example request (create sample role):

```
POST /users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "type": "roles",
        "attributes": {
            "name": "sample role"
        }
    }
}
```

Expected response is `HTTP/1.1 201 OK`, with `application/vnd.api+json` body data representing the role just created.

When role already exists or data is not valid (i.e. data lacks of required fields), POST fails and response is `400 Bad Request - Invalid data`.

Successful response example follows:

```
HTTP/1.1 201 OK
Content-Type: application/vnd.api+json

{
    "data": {
        "id": "7",
        "type": "roles",
        "attributes": {
```

```
            "name": "sample role",
            "created": "2016-12-13T13:38:17+00:00",
            "modified": "2016-12-13T13:38:17+00:00"
        },
        "relationships": {
            "users": {
                "links": {
                    "related": "http://example.com/roles/1/users",
                    "self": "http://example.com/roles/7/relationships/users"
                }
            }
        }
    },
    "links": {
        "self": "http://example.com/roles",
        "home": "http://example.com/home"
    }
}
```

Get role data

You can obtain role data by using `GET /roles` and `GET /roles/(role_id)` endpoint.

`GET /roles`

It returns a collection of roles:

- use `id` query string parameter to retrieve a single role by id
- use `id` query string parameter and `users` token to retrieve users by role id

`GET /roles/` (*role_id*)

Example request (get roles):

```
GET /roles HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "data": [
        {
            "id": 7,
            "type": "roles",
            "attributes": {
                "name": "sample role",
                "description": null,
                "unchangeable": false,
                "created": "2016-12-13T13:38:17+00:00",
                "modified": "2016-12-13T13:38:17+00:00"
            },
            "links": {
                "self": "http://example.com/roles/7"
```

```
            },
            "relationships": {
                "users": {
                    "links": {
                        "related": "http://example.com/roles/7/users",
                        "self": "http://example.com/roles/7/relationships/users"
                    }
                }
            }
        }
    ],
    "links": {
        "self": "http://example.com/roles",
        "home": "http://example.com/home",
        "first": "http://example.com/roles",
        "last": "http://example.com/roles",
        "prev": null,
        "next": null
    },
    "meta": {
        "pagination": {
            "count": 1,
            "page": 1,
            "page_count": 1,
            "page_items": 1,
            "page_size": 20
        }
    }
}
```

GET /roles/ (*role_id*) **/users**

You can obtain role users by using `GET /roles/(role_id)/users` endpoint.

Example request (get users by role 'sample role', id 7):

```
GET /roles/7/users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "data": [
        {
            "id": "19283",
            "type": "users",
            "attributes": {
                "username": "johndoe",
            },
            "links": {
                "self": "http://example.com/users/19283"
            },
            "relationships": {
                "roles": {
                    "links": {
```

```
                        "related": "http://example.com/users/19283/roles",
                        "self": "http://example.com/users/19283/relationships/roles"
                    }
                }
            }
        }
    ],
    "links": {
        "self": "http://example.com/roles/7/users",
        "home": "http://example.com/home",
        "first": "http://example.com/roles/7/users",
        "last": "http://example.com/roles/7/users",
        "prev": null,
        "next": null
    },
    "meta": {
        "pagination": {
            "count": 1,
            "page": 1,
            "page_count": 1,
            "page_items": 1,
            "page_size": 20
        }
    }
}
```

Modify a role

You can modify a role by using `PATCH /roles/(role_id)` endpoint.

`PATCH /roles`

Example request (modify role 'sample role'):

In this example, purpose is modifying 'sample role' name to 'Dummy Role'.

```
PATCH /roles/7 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "id": 7,
        "type": "roles",
        "attributes": {
            "name" : "Dummy Role"
        }
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
```

```
    "data": {
       "id": 7,
        "type": "roles",
        "attributes": {
            "name": "Dummy Role",
            "description": null,
            "unchangeable": false,
            "created": "2016-12-13T13:38:17+00:00",
            "modified": "2016-12-13T14:02:37+00:00"
        },
        "relationships": {
            "users": {
                "links": {
                    "related": "http://example.com/roles/7/users",
                    "self": "http://example.com/roles/7/relationships/users"
                }
            }
        }
    },
    "links": {
        "self": "http://example.com/roles/7",
        "home": "http://example.com/home"
    }
}
```

Remove a role

You can delete a role by using `DEL /roles/(role_id)` endpoint.

`DELETE /roles`

Example request (delete role 'Sample Role', id 7):

```
DELETE /roles/7 HTTP/1.1
Host: example.com
```

Expected response is `204 No Content`. When role is not found, response is `404 Not Found`.

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

Add a user role

You can add a role to a user by using `POST /roles/(role_id)/relationships/users` endpoint. `(role_id)` is a placeholder for role id. You specify user id inside JSON body passed to request.

`POST /roles/` (*role_id*) **`/relationships/users`**

Example request (add role 7 to john doe user, id 19283):

In this example, purpose is adding a role (id 7) to 'johndoe' user (id 19283).

```
POST /roles/7/relationships/users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json
```

```
{
    "data": {
        "type": "users",
        "id": 19283
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "links": {
        "self": "http://example.com/roles/7/relationships/users",
        "home": "http://example.com/home"
    }
}
```

Remove a user role

You can remove a role from a user by using `DELETE /roles/(role_id)/relationships/users` endpoint. `(role_id)` is a placeholder for role id. You specify user id inside JSON body passed to request.

`DELETE /roles/` (*role_id*) **`/relationships/users`**

Example request (remove role 7 from john doe user, id 19283):

In this example, purpose is removing a role (id 7) from 'johndoe' user (id 19283).

```
DELETE /roles/7/relationships/users HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "type": "users",
        "id": 19283
    }
}
```

Response 200 OK is expected.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
    "links": {
        "self": "http://example.com/roles/7/relationships/users",
        "home": "http://example.com/home"
    }
}
```

Objects `/objects`

`/objects` endpoint deals mainly with object retrieval and CRUD operations on objects in general.

Main responsabilities of this endpoint are:

- get a single object
- get a collection of objects, using many filters like trees navigation, search, types
- create a new object
- modify or remove an existing object

Get a single object

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

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": {
      ....
    },
    "method": "get",
    "params": [],
    "url": "https://example.com/api/objects/15"
}
```

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`.

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"
}
```

Administration `/admin`

Note: This endpoint is not yet available

`/admin` endpoint deals with administrative operations on current project

Managed resources by this endpoint are:

- asynchronous jobs

- client applications definitions, including API KEYs management
- custom endpoints
- configuration properties

Object types `/object_types`

`/object_types` endpoint deals with object design and modeling

Main responsabilities of this endpoint are:

- create, update custom object types
- show list fo available object types
- activate or deactivate object types
- create, update properties (core and custom) of an object
- list properties of an object type

Relations `/relations`

Note: This endpoint is not yet available

`/relations` endpoint is used to manage relations between objects

Main responsabilities of this endpoint are:

- create, update/remove a relation between objects
- handle which object types are involved in a relation
- list available relations

Signup `/signup`

`/signup` is the endpoint used to allow users to directly register as project user.

Of course users can be created using *Users /users* endpoint, but with `/signup` you may allow autonomous user registration.

Signup works in two steps:

- signup request: a new user is created, but not activated or verified - therefore the new user may or may not perform some actions in your application
- signup activation: through a UUID hash actual activation and verification of user's email is done

Signup request

This request allows unverified user signup via **POST** that will usually be called anonymously.

Minimum required data are: `username`, `password` and `email`.

An `activation_url` is also required in order to activate the user. An activation email is then sent to the user containing the actual activation URL that will have this form:

```
{activation_url}?uuid={uuid}&redirect_url={redirect_url}
```

Where `{uuid}` is a system generated hash, and `{redirect_url}` is an optional parameter passed in `/signup` request.

In your `activation_url` page you have to read the `uuid` query parameter and then proceed to *Signup activation*.

In this first step a user with status `draft` will be created, user verification job is also added to `async_jobs`: after 24h user verification with provided uuid will expire.

POST /signup

Perform user signup request.

Form Parameters

- **username** – Username of user, must be unique.
- **password** – Password of user.
- **email** – User email, must be unique.
- **activation_url** – Activation URL that will be sent via email.
- **redirect_url** – Optional redirect url that will be added to activation URL as parameter.

Status Codes

- 202 Accepted – Successful user creation. User data will be displayed in response.
- 400 Bad Request – Bad request if username or email have already been used by other users.

Example request: Since this is not a *JSON API* request you MUST use `Content-Type: application/json`

```
POST /signup HTTP/1.1
Host: api.example.com
Accept: application/vnd.api+json
Content-Type: application/json

{
    "username": "johannadoe",
    "password": "j0h4nn4d0e",
    "email": "johannadoe@nowhere.xx"
    "activation_url": "http://myactivationsys.xx?dum=my",
    "redirect_url": "app://xx?dum=my"
}
```

Example response:

Some fields are not displayed for brevity.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
```

```
    "data": {
        "id": "1234",
        "type": "users",
        "attributes": {
            "username": "johannadoe",
            "name": null,
            "surname": null,
            "email": "johannadoe@nowhere.xx",
            "status": "draft",
            "uname": "user-johannadoe",
            "title": null,
            "description": null,
        },
        "meta": {
            "blocked": false,
            "last_login": null,
            "last_login_err": null,
            "num_login_err": 0,
            "verified": null,
            "locked": false,
            "created": "2017-07-20T08:48:25+00:00",
            "modified": "2017-07-20T08:48:25+00:00",
        },
        "relationships": {
            "roles": {
                "links": {
                    "related": "http://api.example.com/users/1234/roles",
                    "self": "http://api.example.com/users/1234/relationships/roles
↪"
                }
            }
        }
    },
    "links": {
        "self": "http://api.example.com/signup",
        "home": "http://api.example.com/home"
    }
}
```


Signup activation

User verification and activation are done via a simple **POST** like in the following example that should be invoked in your **activation url** page after reading the passed **uuid** parameter.

On success an HTTP 202 status code is returned with an empty body.

POST /signup/activation
Perform user signup activation.

Form Parameters

- **uuid** – UUID of signup activation.

Status Codes

- 202 Accepted – Successful activation.
- 404 Not Found – Not found, if provided UUID is incorrect or expired.

Example request: Since this is not a *JSON API* request you MUST use `Content-Type: application/json`

```
POST /signup/activation HTTP/1.1
Content-Type: application/json

{
    "uuid": "96b0b9fe-17fa-4cf8-bffa-1cd506421227"
}
```

Status `/status`

`/status` is a service endpoint to check BEdita service status: environment, software versions, filesystem, database connection are checked.

In normal **production mode** no details are given about software versions and about warnings or failures, whereas in **debug mode** more details are shown.

Only **GET** method is available

Example request:

```
GET /status HTTP/1.1
Accept: application/json
```

Expected response:

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

  {
      "links": {
          "self": "http://example.org/home",
          "home": "http://example.org/home"
      }

      "meta": {
          "status": {
              "environment": "ok"
          }
      }
  }
```

In debug mode more informations will be displayed, including software versions and plugins loaded

Debug mode response:

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

  {
      "links": {
          "self": "http://example.org/home",
          "home": "http://example.org/home"
      }

      "meta": {
```

```
        "status": {
            "environment": "ok",
            "errors": [ ],
            "debug": true,
            "plugins": [
                "BEdita/API",
                "BEdita/Core",
                "DebugKit",
                "Migrations"
            ],
            "versions": {
                "BEdita": "4.0.0-alpha"
            }
        }
    }
}
```


Posters `/posters`

Note: This endpoint is not yet available

`/posters` endpoint gets the thumbnail url of an image representation of an object or a collection of objects.

Main responsabilities of this endpoint are:

- get image representation for an object or a list of objects
- get thumbnail URLs with given parameters (like width and height) for an object or a list of objects

Trash `/trash`

`/trash` endpoint deals with trashcan contents management.

Main responsabilities of this endpoint:

- show trashcan contents
- view trashcan single content
- restore object (remove from trashcan, restore to system)
- delete object (remove from database)

Contents in trashcan

You can obtain trash contents by using `GET /trash` and `GET /trash/(object_id)` endpoint.

`GET /trash/`

`GET /trash` returns response `200 OK` and contents as array, in 'data', as described in following example.

Example request:

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

Example response:

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

{
    "data": [
      ....
    ],
    ....
}
```

GET /trash/ (*object_id*)

`GET /trash/(object_id)` returns response `200 OK` if content is found, `404 Not Found` otherwise.

Example request:

```
GET /trash/154 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

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

{
    "data": {
      ....
    },
    ....
}
```

Restore contents

You can restore contents by using `PATCH /trash/(object_id)` endpoint.

PATCH /trash/ (*object_id*)

Example request (restore object 55920)

In this example, purpose is restoring object 55920.

```
PATCH /trash/55920 HTTP/1.1
Host: example.com
Accept: application/vnd.api+json
Content-Type: application/vnd.api+json

{
    "data": {
        "id": 55920,
        "type": "objects"
```

```
    }
}
```

Response `204 No Content` is expected. When object is not found in trashcan, response is `404 Not Found`.

```
HTTP/1.1 204 No Content
Content-Type: application/vnd.api+json
```

Delete contents

You can completely remove contents from system by using `DELETE /trash/(object_id)` endpoint.

`DELETE /trash/` (*object_id*)

Example request (delete object 55920):

```
DELETE /trash/55920 HTTP/1.1
Host: example.com
```

Expected response is `204 No Content`. When object is not found in trashcan, response is `404 Not Found`.

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

CHAPTER 9

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 wants to access protected resources. This token is sent in `Authorization` HTTP header using a `Bearer` scheme on each request like this: `Authorization: Bearer <token>`

json api

JSON API JSON API is a specification for how a client should request that resources be fetched or modified, and how a server should respond to those requests. JSON API is designed to minimize both the number of requests and the amount of data transmitted between clients and servers. This efficiency is achieved without compromising readability, flexibility, or discoverability [cit. from the specification]

jwt

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 a valid algorithm defined by JSON Web Signature (JWS) specification (for example a shared secret HMAC).

More info here.

object An object in BEdita is the atomic content of your project's data, it could be a core types like a document, an event, an image, a video or it could be a custom type defined specifically in your project. Have a look at *Objects in BEdita* for a detailed overview.

project A project in BEdita is an independent data set consisting mainly of objects, resources and media files; you may think of a database with a set of related resources like media files and configurations; each project will expose its own endpoint to applications

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.

role A role in RBAC permission model (https://en.wikipedia.org/wiki/Role-based_access_control) is used to assign permissions to perform some operations

user Project users accessing resources with credentials to login; main attributes are username and password or some external auth provider identifiers and other profile data like first name, surname, and other contact information; each user has usually at least one role, used to grant access on endpoint operations; user authentication is not always mandatory, some endpoints may respond to GET anonymous requests, but to perform **write** operations a user has to be identified

CHAPTER 10

Indices and tables

- genindex

HTTP Routing Table

/auth

/objects

/roles

/signup

/trash

/users

Index

Symbols

A

B

D

J

O

P

R

S

U