dsb API Client PHP library Documentation

Release 1.0

educa.ch

May 30, 2017

Contents

CHAPTER 1

Introduction

The dsb Client library is a suite of PHP components that facilitate building new applications that communicate with the national catalog of the Swiss digital school library (also knowns as dsb, *Digitale Schulbibliothek*).

This national catalog exposes a RESTful API (more information here), which allows *partners* (organizations that have the right to access the catalog) to write to the catalog, as well as read from it and search for specific *descriptions*. A *description* is a piece of data following the *LOM-CH* standard. The *LOM-CH* standard is a superset of the international LOM standard. *LOM-CH* is fully compatible with *LOM*, but the inverse is not true (*LOM-CH* has more fields).

CHAPTER 2

User Guide

Installation

Using Composer

The easiest way, by far, is to use Composer. Add the following line to your `composer.json` file's `"require"` hash:

```
{
    "require": {
        "educach/dsb-client": "dev-master"
    }
}
```

Call the following command to download the library:

```
composer install
```

After installing, you need to require Composer's autoloader:

```
require 'vendor/autoload.php';
```

Manual installation

If you wish to use this library without using Composer, you can download a release here, or do a checkout using Git at `https://github.com/educach/dsb-client.git`.

Make sure you have some sort of autoloading mechanism in place. The dsb Client library is PSR-4 compatible.

Quickstart

A note on API versions

At the time of writing, the RESTful API is at version 2. It is highly likely that this API will undergo backward incompatible changes in the future, as the LOM-CH standard evolves. In order to plan for this in advance, the dsb Client library has an abstract, base class for client implementations, and version specific implementations that inherit from it.

Make sure to choose the correct client version, depending on which version of the RESTful API your connecting to. For instance, for communicating with the version 2 of the RESTful API, you should use `ClientV2`.

Authentication

All communication with the API requires prior authentication. Upon registering with educa.ch to become a *content partner*, you received a private RSA key, a username (usually an email address) along with a passphrase. You need to pass this information to the client class.

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $client->authenticate();
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

Search

It is possible to search the catalog for specific descriptions. The search is very powerful and flexible, and beyond the scope of this documentation. Refer to the RESTful API documentation for more information.

Search results are not *LOM-CH* objects, but do contain some of their information. The `Educa\DSB\Client\Lom\LomDescriptionSearchResult` class can take a JSON decoded data structure and facilitate it's usage.

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;
use Educa\DSB\Client\Lom\LomDescriptionSearchResult;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $client->authenticate();
    $searchResult = $client->search('Cookies');
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
```

```
}

$results = array();
foreach($searchResult['result'] as $lomData) {
    $results[] = new LomDescriptionSearchResult($lomData);
}

// Get the first LomDescriptionSearchResult.
$lomDescription = $results[0];

// Fetch the field data.
echo $lomDescription->getTitle();
echo $lomDescription->getTeaser();
```

Loading a description

It is also possible to load a full LOM-CH description. This will contain a lot more information than the search results.

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;
use Educa\DSB\Client\Lom\LomDescription;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $client->authenticate();
    $lomDescription = $client->loadDescription('asd789asd9hasd-asd7asdas-asd897asd978
↪');
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

It is possible to fetch LOM-CH field data using special methods:

```
echo $lomDescription->getLomId();
echo $lomDescription->getPreviewImage();
```

Fields that contain data in multiple languages can be instructed to return the information in one language only by specifying a language fallback array. The first language that matches will be returned. If no match is found, the field will be returned in "raw" format (meaning, multilingual fields will be returned as an associative array, with field values keyed by language).

```
// This will first look for a German title, then fallback to French and
// finally Italian.
echo $lomDescription->getTitle(['de', 'fr', 'it']);

// This will look for French first and fallback to English.
echo $lomDescription->getDescription(['fr', 'en']);
```

Not all fields have shortcut methods. For fields that the `Educa\DSB\Client\Lom\LomDescriptionInterface` interface does not define shortcuts for, you can use the `getField()` method. For nested fields, use a *dot* (`.`) notation:

```php
echo $lomDescription->getField('lomId');

// Use a dot (.) notation to fetch nested fields.
echo $lomDescription->getField('lifeCycle.version');

// Fields that are arrays can use numeric field names to get specific items.
echo $lomDescription->getField('technical.keyword.0');

// Fields that are multilingual can use a language fallback array as the
// second parameter.
echo $lomDescription->getField('general.title', ['de', 'fr']);
```

Sending anonymous usage data

It is possible for partners to participate in the effort to provide a better service by sending anonymous user data. This is fully optional, and no data is tracked by default. Applications can send 1 or more of these headers, as they see fit. Applications can send the following HTTP headers:

- `X-DSB-TRACK-ID`: A tracker ID, managed by the client application, to track the user across searches and page loads. This can be a completely arbitrary value, like a random hash. No personal details should be stored in this header.
- `X-DSB-REFERER`: The HTTP *referrer* value.
- `X-DSB-TRACK-IP`: An IP to track the user. **IPs are anonymized by the dsb API**, meaning no personal data is stored. For example, `192.168.1.28` will be stored as `192.168.1.xxx`.

Not all headers are required. Applications can send only 1, 2, or all 3 of them, in any possible combination.

```php
use Educa\DSB\Client\ApiClient\ClientV2;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

// When using a salt to generate a tracker ID, in order for this ID to
// remain consistent across page loads, the salt should be computed only
// once, either per session, per user, or even once for the entire
// application.
$salt = uniqid();
$client->addRequestHeader('X-DSB-TRACK-ID', md5($salt . session_id()));
$client->addRequestHeader('X-DSB-REFERER', $_SERVER['HTTP_REFERER']);
$client->addRequestHeader('X-DSB-TRACK-IP', $_SERVER['REMOTE_ADDR']());

try {
    $client->authenticate();
    // Start making requests, which will send the above usage data.
} catch(\Exception $e) {
    // The request failed.
}
```

Authentication

Almost all communication with the API requires prior authentication. Authentication happens through the exchange of data, signed with a private key. The API, which has access to the *public* key, verifies the validity of this signed data, and returns an *authentication token*. This token then needs to be passed in the request headers for each request.

Getting a private key

You must register with educa.ch to become a *content partner*. You will receive a private RSA key, a username (usually an email address) along with a passphrase. These three elements will be used by the client library to request an authentication token from the API.

Authenticating

Authentication is pretty simple:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $client->authenticate();
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

The Client class will throw an exception in several cases:

- The private key could not be read: check the path, or that the private key is readable.
- The passphrase could not be loaded into memory: make sure the passphrase is correct.
- The request failed: the server did not return a *200* status code. Check the error message.
- The request succeeded, but the response did not contain a token.

If no exception is thrown, the authentication was successful. Once authenticated, the class can communicate with the API.

Chaining methods

The client class supports chaining methods. It is thus possible to chain an authentication with another action. For example:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $client->authenticate()->search('Dog');
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
} catch(ClientRequestException $e) {
    // The request failed.
}
```

LOM-CH Descriptions

A *description* is a piece of data following the *LOM-CH* standard. The *LOM-CH* standard is a superset of the international LOM standard. *LOM* stands for *Learning Object Metadata*. It is a standard for representing learning resources, be it books, videos, websites, games, etc. The dsb (*Digitale Schulbibliothek*) groups many of these descriptions in a large catalog, with an API for searching for descriptions, loading a full description, or even adding new ones.

All communication with the API regarding descriptions requires prior authentication. See *Authentication* for more information.

Searching descriptions

Full text search

It is possible to search for description in a variety of ways. The most obvious is doing a full text search:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies');
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This will do a full text search on the word *Cookies*. All descriptions that fit the bill are returned by the API in JSON. The client will convert this JSON to an associative array before returning it. The returned value has 3 keys:

- *numFound*: The number total number of results the API found. This could be more than the number of actual results returned (defaults to 50).
- *result*: An array of descriptions, the actual search results.
- *facets*: A tree of facet data. More on this later. By default, this is empty.

You can get this data like so:

```
// Fetch the descriptions.
$descriptions = $searchResult['result'];
```

By default, each description has the following properties:

- *lomId*: The description identifier.
- *title*: The title of the description.
- *teaser*: A shorter version of the description body text.
- *previewImage*: An image representing the learning resource.

Other fields can be added if required. More below.

Adding facets

The API supports something called *Faceted Search*. Simply put, this allows a search engine to dynamically tell what kind of filtering would be possible for the returned results. This information is then usually used to display some sort of list of options (like checkboxes) which can be displayed on a search form to allow dynamic filtering of the results.

By default, no facets are active, but you can ask the API to compute facets for the current query:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', ['learningResourceType
↪']);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This will compute a list of *resource types* (*text*, *image*, *website*, etc) that are available for all found results. You may use this information to build a search form, and display these facets as checkboxes. The values of these checkboxes can then be used as *filters* (more below).

It is possible to pass more than one facet to the Client:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', ['learningResourceType
↪', 'educaSchoolLevels']);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

A full list of available facets can be found here. A live-example of how these facets can be used can be found here.

Filtering results

It is possible to add filters to narrow the search down. This is often closely related to *facets* (see above). A *filter* is an object, where each property name is a filter name, and its value is an array of possible values. For example, imagine we only want descriptions in German:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;
```

```
$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], ['language' => ['de
↪']]);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This will filter all results and only show ones in German. Multi-value filters are possible as well. Multiple values are treated as *OR*, not *AND*:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], [
↪'learningResourceType' => ['text', 'image']]);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This will filter by descriptions that are either text-based or image-based (or both).

Additional fields

It is possible to add more fields to the search results. The 4th parameters passed to the client class when searching allows you to specify what more fields should be returned for each search result. For example, the following would add the `language` and `version` properties to the result:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], [], ['language',
↪'version']);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

Read the API documentation for more information.

Pagination and limiting the number of results

It is possible to offset the results, effectively giving applications a way to support *pagination*. The offset is the 5th parameter, and represents by how many items the results should be offset (usually a multiple of the 6th parameter, *limit*; more below). The following will show results 21 to 70 (50 being the default *limit*):

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], [], [], 20);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

It is also possible to limit the number of results. The following will only show 20 results (instead of 50, the default):

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], [], [], 0, 20);
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

Manipulating results

Manipulating this search data might prove cumbersome. This is why there is a special class, called `LomDescriptionSearchResult`, which can greatly simplify displaying search results. Simply pass the JSON-decoded value to the constructor:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;
use Educa\DSB\Client\Lom\LomDescriptionSearchResult;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $searchResult = $client->authenticate()->search('Cookies', [], [], ['language',
↪'version']);
} catch(ClientRequestException $e) {
    // The request failed.
```

```
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}

foreach($searchResult['result'] as $lomData) {
    $lomDescription = new LomDescriptionSearchResult($lomData);

    echo $lomDescription->getTitle();
    echo $lomDescription->getTeaser();
    echo $lomDescription->getLomId();
    echo $lomDescription->getPreviewImage();
}
```

For additional fields, like `language` and `version` in our example, you may use the method `getField()`. This method takes a field name as a parameter:

```
foreach($searchResult['result'] as $lomData) {
    $lomDescription = new LomDescriptionSearchResult($lomData);

    echo $lomDescription->getField('language');
    echo $lomDescription->getField('version');
}
```

Of course, this also works for the default fields:

```
foreach($searchResult['result'] as $lomData) {
    $lomDescription = new LomDescriptionSearchResult($lomData);

    echo $lomDescription->getField('title');
    echo $lomDescription->getField('teaser');
    echo $lomDescription->getField('lomId');
    echo $lomDescription->getField('previewImage');
}
```

Loading a description

It is possible to load the full data for a resource. This will contain all meta-data, as well as data from the Ontology server.

Ontology data

Ontology data provides human-readable strings for *vocabulary* entries. For example, a description can have several *contributors*. Each of these contributors has a *role*, like *author*, *editor*, etc. These are *machine-readable* names, and are always the same, regardless of which language the description is in. In order to keep the human-readable values, as well as translations, of these vocabulary entries centralized, one can query the *Ontology Server*. This can be done directly through the API. See *Ontology data* for more information. However, the API "injects" most if this data directly into the loaded descriptions, which saves us the hassle.

Multilingual descriptions

Because this is communicating with the *Swiss* national catalog (which has 4 official languages), many descriptions are multilingual. When loading a description, many fields, like *title*, *keyword*, etc, can have different values, one per language.

Loading a description

Loading a description requires knowing its *LOM identifier*. This is a UUID, or a MD5 hash prefixed with *archibald###* for older versions.

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $descriptionData = $client->authenticate()->loadDescription('asd89iowqe-sadjqw98-
↪asd87a9doiiuowqe');
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This loads the description data as an associative array into `$descriptionData`. Look at the API documentation for more information on this data structure.

Manipulating a description

This object can be pretty hard to manipulate. That is where `LomDescription` comes in. The `LomDescription` class can take a JSON-decoded LOM-CH data object and expose its properties in a much more convenient way:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;
use Educa\DSB\Client\Lom\LomDescription;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $descriptionData = $client->authenticate()->loadDescription('asd89iowqe-sadjqw98-
↪asd87a9doiiuowqe');
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}

$lomDescription = new LomDescription($descriptionData);

echo $lomDescription->getTitle();
echo $lomDescription->getDescription();
echo $lomDescription->getLomId();
echo $lomDescription->getPreviewImage();
```

Fields that contain data in multiple languages can be instructed to return the information in one language only by

specifying a language fallback array. The first language that matches will be returned. If no match is found, the field will be returned in "raw" format (meaning, multilingual fields will be returned as an associative array, with field values keyed by language).

```
// This will first look for a German title, then fallback to French and
// finally Italian.
echo $lomDescription->getTitle(['de', 'fr', 'it']);

// This will look for French first and fallback to English.
echo $lomDescription->getDescription(['fr', 'en']);
```

Not all fields have shortcut methods. For fields that the `LomDescriptionInterface` interface does not define shortcuts for, you can use the `getField()` method. For nested fields, use a *dot* (`.`) notation:

```
echo $lomDescription->getField('lomId');

// Use a dot (.) notation to fetch nested fields.
echo $lomDescription->getField('lifeCycle.version');

// Fields that are arrays can use numeric field names to get specific items.
echo $lomDescription->getField('technical.keyword.0');

// Fields that are multilingual can use a language fallback array as the
// second parameter.
echo $lomDescription->getField('general.title', ['de', 'fr']);
```

Creating a new description

todo

Updating a description

todo

Validating a description

It is possible to validate a description to check if no mandatory fields are missing, that they are well formed and respect the LOM-CH standard. Look at the API documentation for more information on this data structure.

Simple validation script:

```
$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

// Load a json file containing the LOM object
$f = 'lom_object.json';
try {
    $json = file_get_contents($f);
    $result = $client->authenticate()->validateDescription($json);
    echo "Using file $f\n";
    if (!empty($response['valid'])) {
        echo "\n> Description is valid.";
    } else {
        echo "\n> Description is invalid.\n";
```

```
        echo "\nServer response:\n";
        echo "==============================\n";
        print_r($response);
        echo "==============================\n";
    }
} catch(ClientRequestException $e) {
    // The request failed.
    print "The post request failed. (" . $e->getMessage() . ')';
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
    print "The authentification failed. (" . $e->getMessage() . ')';
}
echo "\n";
```

Response syntax

The response will always contain a `valid` key, which is a boolean.

If the submitted LOM object is invalid, the `errors` key will be populated with a list of issues.

In case of an invalid LOM object, the API will return:

```
{"valid":false,"message":"Description is not complete or not compliant.","errors":{
↪"general.identifier":"missing","general.description":"missing","general.language":
↪"missing"}}
```

The client returns a JSON decoded array:

```
Array
(
    [valid] =>
    [message] => Description is not complete or not compliant.
    [errors] => Array
        (
            [general.identifier] => missing
            [general.description] => missing
            [general.language] => missing
        )

)
```

Files

Files can be uploaded to the dsb (*Digitale Schulbibliothek*) API server to be hosted there. The dsb API provides methods for images to be resized on the fly. This is a functionality *partners* can leverage to display images of certain sizes to their users, instead of using the original, high-resolution version. Check out the official dsb API documentation for more information.

Uploading files

Files can be uploaded using the `uploadFile` method. At the time of writing, the dsb API only allows *image files*. Make sure you upload an image in a supported format.

```php
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $result = $client->authenticate()->fileUpload('/path/to/file.jpg');

    // The result contains the URL at which the file is accessible.
    echo $result['fileUrl'];
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
} catch(\RuntimeException $e) {
    // Either the file is not readable or doesn't exist.
}
```

Ontology data

todo

Curricula

A curriculum allows the national catalog to categorize and describe a resource in context of a specific course curriculum. For example, in the French part of Switzerland, many schools try to adhere to the PER curriculum. LOM descriptions can contain meta-data information about how it can apply to this curriculum.

Curricula can vary greatly, and there's not often a common *ground* or standard that they all can refer to. Instead, it is up to the application to know how to treat the curriculum at hand.

In an effort to provide some level of abstraction and code re-use, the dsb Client Library comes with a set of curricula implementations. These follow a very simple, yet powerful logic of storing curricula *terms* (e.g., *school levels*, *contexts*, *themes*, *objectives*, etc) in a flat tree structure. This allows applications to navigate the tree and get meta-data information about each element.

Abstraction and developing new curriculum implementations

The base classes and interfaces make no assumption about the actual data format of the curricula. For instance, the official document for the PER curriculum data structure is an XML file. The official document for the *standard* (a.k.a. *educa*) curriculum data structure is a JSON file, and so on.

However, all curricula implementations *must* implement the `CurriculumInterface` interface. This interface describes methods for fetching curricula meta-data and structural information, allowing applications to better understand how a typical curriculum tree could look.

For example, the `describeDataStructure()` method returns a standard format of describing relationships and types of curriculum *terms*.

```php
use Educa\DSB\Client\Curriculum\EducaCurriculum;

$curriculum = new EducaCurriculum();

var_export($curriculum->describeDataStructure());
// Results in:
// array (
//   0 =>
//   stdClass::__set_state(array(
//      'type' => 'educa_school_levels',
//      'childTypes' =>
//     array (
//       0 => 'context',
//     ),
//   )),
//   1 =>
//   stdClass::__set_state(array(
//      'type' => 'context',
//      'childTypes' =>
//     array (
//       0 => 'school_level',
//     ),
//   )),
//   2 =>
//   stdClass::__set_state(array(
//      'type' => 'school_level',
//      'childTypes' =>
//     array (
//       0 => 'school_level',
//     ),
//   )),
//   3 =>
//   stdClass::__set_state(array(
//      'type' => 'educa_school_subjects',
//      'childTypes' =>
//     array (
//       0 => 'discipline',
//     ),
//   )),
//   4 =>
//   stdClass::__set_state(array(
//      'type' => 'discipline',
//      'childTypes' =>
//     array (
//       0 => 'discipline',
//     ),
//   )),
// )
```

`describeTermTypes()` provides even more information on what the term *types* actually stand for.

```php
use Educa\DSB\Client\Curriculum\EducaCurriculum;

$curriculum = new EducaCurriculum();

var_export($curriculum->describeTermTypes());
// Results in:
// array (
```

```
//   0 =>
//   stdClass::__set_state(array(
//      'type' => 'context',
//      'purpose' =>
//     array (
//       'LOM-CHv1.2' => 'educational level',
//     ),
//   )),
//   1 =>
//   stdClass::__set_state(array(
//      'type' => 'school level',
//      'purpose' =>
//     array (
//       'LOM-CHv1.2' => 'educational level',
//     ),
//   )),
//   2 =>
//   stdClass::__set_state(array(
//      'type' => 'discipline',
//      'purpose' =>
//     array (
//       'LOM-CHv1.2' => 'discipline',
//     ),
//   )),
// );
```

`asciiDump()` provides a way to dump a tree representation to a ASCII string, helping in debugging.

```
use Educa\DSB\Client\Curriculum\EducaCurriculum;

$curriculum = new EducaCurriculum();

// Do some treatment, constructing the curriculum tree...

print $curriculum->asciiDump();
// Results in:
// --- root:root
//     +-- context:compulsory education
//         +-- school level:cycle_3
//             +-- discipline:languages
//                 +-- discipline:french school language
//     +-- context:special_needs_education
//         +-- discipline:languages
//             +-- discipline:french school language
```

The standard, static `createFromData()` method provides a standard factory method for creating new curriculum elements, although the format of the actual data passed to the method is completely left to the implementor.

A curriculum tree consists of `TermInterface` elements. Each element has the following methods, allowing applications to navigate the tree:

- `hasChildren()`: Whether the term has child terms.
- `getChildren()`: Get the child terms.
- `hasParent()`: Whether the term has a parent term.
- `getParent()`: Get the parent term.
- `isRoot()`: Whether the term is the root parent term.

- `getRoot()`: Get the root parent term.
- `hasPrevSibling()`: Whether the term has a sibling term "in front" of it.
- `getPrevSibling()`: Get the sibling term "in front" of it.
- `hasNextSibling()`: Whether the term has a sibling term "after" it.
- `getNextSibling()`: Get the sibling term "after" it.
- `findChildByIdentifier()`: Allows to search direct descendants for a specific term via its identifier.
- `findChildByIdentifierRecursive()`: Same as above, but recursively descends onto child terms as well
- `findChildrenByName()`: Allows to search direct descendants for a specific term via its name.
- `findChildrenByNameRecursive()`: Same as above, but recursively descends onto child terms as well.
- `findChildrenByType()`: Allows to search direct descendants for a specific term via its type.
- `findChildrenByTypeRecursive()`: Same as above, but recursively descends onto child terms as well.

Furthermore, it has one more method, `describe()`, which allows applications to understand what kind of term they're dealing with.

Thanks to these methods, applications can navigate the entire tree structure and treat it as a flat structure.

There is a basic implementation for terms, `BaseTerm`. It also implements the `EditableTermInterface` interface, and is usually recommended for use within any curriculum implementation.

"Standard" (educa) curriculum

This curriculum has been deprecated, and replaced by the Classification System

The "standard" (or *educa*) curriculum is a non-official curriculum that aims to provide some basic curriculum that all Swiss cantons can more or less relate to. Its definition can be found here.

The definition file is a JSON file that can be downloaded from the site (link). The `EducaCurriculum` class can parse this information for re-use. The reason this raw definition data does not *have* to be passed to `EducaCurriculum` every time is that applications might want to cache the parsing result, and pass the cached data in future calls. This can save time, as the parsing can be quite time-consuming and memory intensive.

```
use Educa\DSB\Client\Curriculum\EducaCurriculum;

// $json contains the official curriculum data in JSON format.
$json = file_get_contents('/path/to/curriculum.json');
$curriculum = EducaCurriculum::createFromData($json);

// We can also simply parse it, and cache $data for future use.
$data = EducaCurriculum::parseCurriculumJson($json);

// Demonstration of re-use of cached data.
$curriculum = new EducaCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);
```

The curriculum class supports the handling of LOM *classification* field data (field no 9). This is represented as a series of *taxonomy paths*. Please refer to the REST API documentation for more information. By default, it only considers *discipline* taxonomy paths. If you wish to parse a taxonomy path with another *purpose* key, pass it as the second parameter to `setTreeBasedOnTaxonPath()`.

```php
use Educa\DSB\Client\Curriculum\EducaCurriculum;

// Re-use cached data for the dictionary and curriculum definition.
// See previous example.
$curriculum = new EducaCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);

// $paths is an array of taxonomy paths. See official REST API documentation
// for more info.
$curriculum->setTreeBasedOnTaxonPath($paths);

print $curriculum->asciiDump();
// Results in:
// --- root:root
//     +-- context:compulsory education
//         +-- school level:cycle_3
//             +-- discipline:languages
//                 +-- discipline:german school language
//             +-- discipline:social and human sciences
//                 +-- discipline:citizenship
//                 +-- discipline:history
//     +-- context:post compulsory education
//         +-- discipline:languages
//             +-- discipline:german school language
//         +-- discipline:social and human sciences
//             +-- discipline:history
//             +-- discipline:psychology
//             +-- discipline:philosophy
//         +-- discipline:general_education
//             +-- discipline:identity
```

Of course, you can call `getTree()` to get the root item of the tree, and navigate it.

Classification System

Not a true curriculum, this system allows educational resources to be classified independently from cantonal curricula. Its definition can be found here.

The definition file is a JSON file that can be downloaded from the site (link). The `ClassificationSystemCurriculum` class can parse this information for re-use. The reason this raw definition data does not *have* to be passed to `ClassificationSystemCurriculum` every time is that applications might want to cache the parsing result, and pass the cached data in future calls. This can save time, as the parsing can be quite time-consuming and memory intensive.

```php
use Educa\DSB\Client\Curriculum\ClassificationSystemCurriculum;

// $json contains the official curriculum data in JSON format.
$json = file_get_contents('/path/to/curriculum.json');
$curriculum = ClassificationSystemCurriculum::createFromData($json);

// We can also simply parse it, and cache $data for future use.
$data = ClassificationSystemCurriculum::parseCurriculumJson($json);

// Demonstration of re-use of cached data.
$curriculum = new ClassificationSystemCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);
```

The curriculum class supports the handling of LOM *classification* field data (field no 9). This is represented as a series of *taxonomy paths*. Please refer to the REST API documentation for more information. By default, it only considers *discipline* taxonomy paths. If you wish to parse a taxonomy path with another *purpose* key, pass it as the second parameter to `setTreeBasedOnTaxonPath()`.

```
use Educa\DSB\Client\Curriculum\ClassificationSystemCurriculum;

// Re-use cached data for the dictionary and curriculum definition.
// See previous example.
$curriculum = new ClassificationSystemCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);

// $paths is an array of taxonomy paths. See official REST API documentation
// for more info.
$curriculum->setTreeBasedOnTaxonPath($paths);

print $curriculum->asciiDump();
// Results in:
// --- root:root
//     +-- context:compulsory education
//         +-- school level:cycle 3
//             +-- discipline:languages
//                 +-- discipline:german school language
//             +-- discipline:social and human sciences
//                 +-- discipline:history
//     +-- context:post compulsory education
//         +-- discipline:languages
//             +-- discipline:german school language
//         +-- discipline:social and human sciences
//             +-- discipline:history
//             +-- discipline:psychology
//             +-- discipline:philosophy
//         +-- discipline:general education
//             +-- discipline:identity
```

Of course, you can call `getTree()` to get the root item of the tree, and navigate it.

Plan d'études Romand (PER) curriculum

The *Plan d'études romand* (or *per*) curriculum is an official curriculum for the French speaking cantons in Switzerland. More information can be found here.

The definition data can be fetched via an API, which is openly accessible here. The `PerCurriculum` class can fetch and parse this information for re-use. The reason this data does not *have* to be loaded by `PerCurriculum` every time is that applications might want to cache the parsing result, and pass the cached data in future calls. This can save time, as the parsing can be very time-consuming and memory intensive (it requires hundreds of `GET` requests to the REST API).

```
use Educa\DSB\Client\Curriculum\PerCurriculum;

// $url contains the path to the REST API the class must use.
$url = 'http://bdper.plandetudes.ch/api/v1/';
$curriculum = PerCurriculum::createFromData($url);

// We can also simply parse it, and cache $data for future use.
$data = PerCurriculum::fetchCurriculumData($url);
```

```
// Demonstration of re-use of cached data.
$curriculum = new PerCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);
```

The curriculum class supports the handling of LOM-CH *curriculum* field data (field no 10). This is represented as a series of *taxonomy trees*. Please refer to the REST API documentation, for more information on the structure.

```
use Educa\DSB\Client\Curriculum\PerCurriculum;

// Re-use cached data for the dictionary and curriculum definition.
// See previous example for more info.
$curriculum = new PerCurriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);

// $trees is an array of taxonomy trees. See official REST API documentation
// for more info.
$curriculum->setTreeBasedOnTaxonTree($trees);

print $curriculum->asciiDump();
// Results in:
// --- root:root
//     +-- cycle:1
//         +-- domaine:4
//             +-- discipline:13
//                 +-- objectif:76
//             +-- discipline:11
//                 +-- objectif:77
```

Of course, you can call `getTree()` to get the root item of the tree, and navigate it.

A curriculum tree consists of `TermInterface` elements, just as for the other curricula implementations. However, `PerCurriculum` uses a custom term implementation, `PerTerm`. This implements the same interfaces, so can be used in exactly the same ways as the standard terms. The difference is `PerTerm` exposes a few more methods:

- `findChildByCode()`: Allows to search direct descendants for a specific term via its code (mostly applies to *Objectifs*)
- `findChildByCodeRecursive()`: Same as above, but recursively descends onto child terms as well
- `getUrl()` and `setUrl()`: Get/set the URL property of an item (mostly applies to *Objectifs*)
- `getCode()` and `setCode()`: Get/set the code property of an item (mostly applies to *Objectifs*)
- `getSchoolYears()` and `setSchoolYears()`: Get/set the school years property of an item (mostly applies to *Objectifs* and *Progressions d'apprentissage*)

Lehrplan 21 (lp21) curriculum

The *Lehrplan 21* (or *lp21*) curriculum is an official curriculum for the German speaking cantons in Switzerland. More information can be found here.

The definition file is a XML file that can be downloaded from the site. The `LP21Curriculum` class can parse this information for re-use. The reason this data does not *have* to be passed to `LP21Curriculum` every time is that applications might want to cache the parsing result, and pass the cached data in future calls. This can save time, as the parsing can be very time-consuming and memory intensive (the XML is over 20Mb in size).

```
use Educa\DSB\Client\Curriculum\LP21Curriculum;
```

```php
// $xml contains the official curriculum data in XML format.
$xml = file_get_contents('/path/to/curriculum.xml');
$curriculum = LP21Curriculum::createFromData($xml);

// We can also simply parse it, and cache $data for future use.
$data = LP21Curriculum::parseCurriculumXml($xml);

// Demonstration of re-use of cached data.
$curriculum = new LP21Curriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);
```

The curriculum class supports the handling of LOM-CH *curriculum* field data (field no 10). This is represented as a series of *taxonomy trees*. Please refer to the REST API documentation, for more information on the structure.

```php
use Educa\DSB\Client\Curriculum\LP21Curriculum;

// Re-use cached data for the dictionary and curriculum definition.
// See previous example for more info.
$curriculum = new LP21Curriculum($data->curriculum);
$curriculum->setCurriculumDictionary($data->dictionary);

// $trees is an array of taxonomy trees. See official REST API documentation
// for more info.
$curriculum->setTreeBasedOnTaxonTree($trees);

print $curriculum->asciiDump();
// Results in:
// --- root:root
//     +-- zyklus:3
//         +-- fachbereich:010fby8NKE8fCB79TRL69VS8VT4HnuHmN
//             +-- fach:010ffPWHRKNUdFDK9LRgFbPDLXwTxa4bw
//         +-- fachbereich:010fbNpVqv9R3TePRnCeZECuB4ucv6rEw
//             +-- kompetenzbereich:010kbAnkUn9X9c8kz25FN9zFTFaHdAbPb
//                 +-- handlungs_themenaspekt:010hafG6hGk8FZJWduaNDBGE4zhRWnvXK
//         +-- fachbereich:010fbNpVqv9R3TePRnCeZECuB4ucv6rEw
//             +-- kompetenzbereich:010kbAnkUn9X9c8kz25FN9zFTFaHdAbPb
//         +-- fachbereich:010fbNpVqv9R3TePRnCeZECuB4ucv6rEw
//             +-- kompetenzbereich:010kbAnkUn9X9c8kz25FN9zFTFaHdAbPb
//                 +-- handlungs_themenaspekt:010ha4HnxH3GG5f5mqe8bddWtJK8bbVmD
```

Of course, you can call `getTree()` to get the root item of the tree, and navigate it.

A curriculum tree consists of `TermInterface` elements, just as for the other curricula implementations. However, `LP21Curriculum` uses a custom term implementation, `LP21Term`. This implements the same interfaces, so can be used in exactly the same ways as the standard terms. The difference is `LP21Term` exposes a few more methods:

- `findChildByCode()`: Allows to search direct descendants for a specific term via its code
- `findChildByCodeRecursive()`: Same as above, but recursively descends onto child terms as well
- `getUrl()` and `setUrl()`: Get/set the URL property of an item (mostly applies to *Kompetenzstufe*)
- `getCode()` and `setCode()`: Get/set the code property of an item
- `getVersion()` and `setVersion()`: Get the version of the Lehrplan this item is meant for (mostly applies to *Kompetenzstufe*)
- `getCantons()` and `setCantons()`: Get the *Cantons* this item is meant for (mostly applies to *Fachbereiche*)

- `getCycles()` and `setCycles()`: Get the *cycles* this item applies to (mostly applies to *Kompetenzstufe*)

Mapping between curricula

It is possible to map certain *terms* from one curriculum to another. Not all curricula support being mapped to, or from. Check the REST API documentation for more information.

Mapping is achieved by passing the *source* curriculum ID, the *target* curriculum ID, and the ID of the term to map:

```
use Educa\DSB\Client\ApiClient\ClientV2;
use Educa\DSB\Client\ApiClient\ClientAuthenticationException;
use Educa\DSB\Client\ApiClient\ClientRequestException;

$client = new ClientV2('https://dsb-api.educa.ch/v2', 'user@site.com', '/path/to/
↪privatekey.pem', 'passphrase');

try {
    $suggestions = $client->authenticate()->getCurriculaMappingSuggestions(
        'per',
        'classification_system',
        'objectifs-1'
    );
} catch(ClientRequestException $e) {
    // The request failed.
} catch(ClientAuthenticationException $e) {
    // The authentication failed.
}
```

This will return a list, keyed by term identifier, each containing a list of suggestions.

Unit testing your own application

It is possible to write unit tests for your own application by using the `TestClient` class instead of one of the real implementations. This class implements the same `ClientInterface` interface, and returns mocked results. It's implementation is pretty simple, but can do the trick for many test cases. If you wish to have more complex results, you may simply extend it and implement your own.

It is a good idea to use some kind of *dependency injection* in your application. Symfony and Silex provide such mechanisms out of the box. Another popular method is using Pimple. In the official dsb Client Drupal module, a simple function returns an instance of the client based on the system settings. Anyway, if you make sure the client class is returned by some sort of service container or function, you can make sure it returns a `TestClient` in your testing environment, allowing your unit tests to run without requiring actual access to the API.

It is recommended to look at the source code of the `TestClient`, to get a better understanding of how it can be used inside a test environment.