V2 API

This document describes how to use version 2 of the Bright API.

The Bright API provides programmatic access to a variety of functions provided by the Bright platform, most significantly read and write access to Bright database. Via the API it is possible to register users, assign courses, generate launch URLs, and query results.

This document details how to get connected to the Bright Server, how to authenticate yourself, and what functions are available via which API calls. It also covers what to do when things go wrong, and how to get more information.

More information about Bright and the Bright platform can be accessed from the Aura Homepage.

General Usage

The Bright API is used to interact with Bright Server. Typical things you might do with Bright are

- access data about learners, courses, registrations.
- You might be querying data or pushing data in.

Regardless, all Bright API calls function in the same way. Fundamentally, the Bright API is a RESTful API, so much of what is covered here should seem familiar if you've worked with a restful API before. If not, you may want to familiarize yourself with the concepts here: http://en.wikipedia.org/wiki/Representational*state*transfer

> **Representational State Transfer** (**REST**) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant Web service design model.
>
> The term *representational state transfer* was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.[1][2] Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.[3][4]
>
> Conforming to the REST constraints is generally referred to as being "RESTful".[5]

Source: Wikipedia

RESTful API

The Restful api of bright is built around our main resource types:

- courses
- registrations
- users
- api keys
- invitations

for each resource type, we follow the following model, where a specific path, http verb and action are consistent depending on the bright resource you are accessing.

HTTP Verb	Path	action	used for
GET	/{{resource}}	index	display a list of all {{resource}}
GET	/{{resource}}/new	new	return an HTML form for creating a new {{resource}}
POST	/{{resource}}	create	create a new {{resource}}
GET	/{{resource}}/gcreate	create	create a new {{resource}} with a GET request. This is an alias to the POST get request provided to allow the {{resource} to be created from a web browser without using a POST. A variety of cross-domain scripting issues can be addressed using the GET create alias (gcreate).
GET	/{{resource}}/:id	show	display a specific {{resource}}

GET	/{{resource}}/:id/edit	edit	return an HTML form for editing a {{resource}}
PUT	/{{resource}}/:id	update	update a specific {{resource}}
GET	/{{resource}}/gupdate/:id	update	update a specific {{resource}} via a GET request. See the comment on gcreate above.
DELETE	/{{resource}}/:id	destroy	delete a specific {{resource}}

List of Resouces Available

- Bright Api Keys (/bright/api/v2/api_key)
- Course Provider (/bright/api/v2/course_provider)
- Courses (/bright/api/v2/course)
- Registrations (/bright/api/v2/registation)
- Realm User (/bright/api/v2/realm_user)
- Embed Templates (/bright/api/v2/embed_template) [Note: in v3, this will be renamed simply "template"].
- Invitations (/bright/api/v2/invitation)

Special Considerations for Cross-Browser Scripting Restrictions

One thing about the RESTFul architecture is does not interoperate well with cross browser scripting restrictions in modern browsers. Specifically, you are probably aware that you can not POST or PUT data via JSONP which means you can interact with a *pure* RESTFul API from a web page delivered from a web server other the one hosting the Bright API.

To alleviate this problem, we provide two "GET"-oriented aliases, gcreate and gupdate, that work identically to create and update, and are available via HTTP GET calls.

Quick Start

One of the nice things about the Bright API is its easy to get started. First off, you need a few things:

Prerequisite Data

- Your Bright API URL. Don't know it? Ask us at support@aura-software.com!

If you are using a SCORMCloud course provider, API access can be provided using your SCORMCloud APP ID and secret key. These are available from Aura Support, or from the SCORMCloud console available at https://cloud.scorm.com .

- The SCORMCloud APP ID for your SCORMCloud Application.
- The SCORMCloud Secret Key for your SCORMCloud Application.

You can also access Bright with a Bright realm app id and key (expand) (#access-modes/via-realm-id-and-secret-key).

Certain functionality can only be accessed with a realm key, such as invitations or the utilization of multiple course providers.

For the purposes of this example, we will use the following:

- Bright API URL: http://[BRIGHT URL]/bright/api/v2
- SCORMCloud APPID: **HN0CR2W8J8**
- SCORMCloud Secret Key: **nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N**

IF we know this, we can already use our API! To do so, we can test this straight from curl. Curl is a command line tool, and it is easy to install on most systems.

When learning the Bright API, we recommend you start by assembling some simple curl commands from your command line in order to get a feel for what is possible.

So let's get our list of course from our API:

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.xml?sc_app_id=HN0CR2W8J8&
sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N'
```

If you've executed this correctly, you'll get a result like:

And the result:

```
<?xml version="1.0" encoding="UTF-8"?>
<scorm-cloud-courses type="array">
  <scorm-cloud-course>
    <course-provider-id type="integer">6</course-provider-id>
    <created-at type="datetime">2012-11-26T12:10:40Z</created-at>
    <custom></custom>
    <id type="integer">184</id>
```

```
      <metadata>{"title":"ENT Foundation - Post Training QUIZ",
                    "description":"",
                                "duration":"0",
                                "typicaltime":"0",
                                "keywords":null}
          </metadata>
      <registration-count type="integer">1</registration-count>
      <sc-course-id>1-507727747154e</sc-course-id>
      <size type="integer">157758</size>
      <title>System Test Course</title>
      <updated-at type="datetime">2013-01-17T16:20:10Z</updated-at>
      <versions type="integer">-1</versions>
    </scorm-cloud-course>
  </scorm-cloud-courses>

```

Specifying a results format.

Fetching an XML result

To fetch your results in XML format, append a '.xml' to the url, *before* the request parameters. For example

```
http://[BRIGHT URL]/bright/api/v2/course.xml
```

Fetching a JSON result

Let's say if you are using the API from Javascript, and you'd like your results back as JSON. Easy, just rewrite the URL to use 'course.json' instead of 'course.xml'

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?
sc_app_id=HN0CR2W8J8&
sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N'
```

And the result (formatted for readibility:

```
[
 {
        "created_at":"2012-11-26T12:10:44Z",
        "id":187,
        "registrations":2,
        "sc_course_id":"1-5098f07394cbd",
        "course_provider_id":6,
        "size":1599415,
        "title":"Proktologie",
        "updated_at":"2012-11-27T22:30:40Z",
        "versions":-1
 },
 {
        "created_at":"2012-11-29T16:11:53Z",
        "id":200,
        "registrations":1,
        "sc_course_id":"1-50b63e903dd43",
        "course_provider_id":6,
        "size":2211931,
        "title":"Proktologie",
        "updated_at":"2012-12-07T15:48:43Z",
        "versions":-1
 }
]
```

Access Modes

The Bright API can be accessed to two different ways:

- SCORMCloud App ID and Secret Key
- Bright API Key
- [Brigh Realm GUID and Secret Key) (#access-modes/realm-guid-and-secret-key)

If you do not specify either the api key/secret key pair **OR** the Bright API key, you will receive HTTP code 401 (Unauthorized) in your response.

If you specify multiple authentication models, you will received a 501 (not implemented).

If you specify an unknown SCORMCloud APPId and Secret key, you will receive a 500 (server error).

Via SCORMCloud App ID and Secret Key

In all of the examples so far, we've been using direct access to the API using the SCORMCloud secret key and app id. Generally speaking, this is fine for server side code that is secured and publicly accessible. You should NEVER share the APP ID and secret key since this give complete access to all of your data.

Example:

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?sc_app_id=\
HN0CR2W8J8&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N'
```

Via Bright API Key

If you want to access the Bright API from some browser side Javascript, using the SCORMCloud or Bright Realm secret key is no good. NEVER put your Secret Key into the browser via Javascript.

Instead the Bright API allows you to create an authentication token that you can use to send to the browser. When you generate this token, typically you specify

- the SCORMCloud app id/secret key OR bright realm secret key
- optionally the user

These application tokens are disabled after a short period of time. Do not hard code the use of a Bright API key, as these keys expire after a period of time.

Here's an example of getting an API token. This first call creates a special access token to a specific user for a specific SCORMCloud application. The key that is returned is suitable for embedding in a web page for use by browser side Javascript.

```
curl 'http://[BRIGHT URL]/bright/api/v2/api_key/create?sc_app_id=HN0CR2W8J8
&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N
&user_email=admin@aura-software.com'
(returns) df8d1350a6b31378a86b967767f4bba1
```

You can now omit the secret key and app id from subsequent calls and just use the API key:

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.xml?\
api_token=bdb273e9cdace9698c34d97070cb392d'
```

This API token is now "bound" to the access level specified when the key was created.

Via Realm ID and Secret Key

TODO

API Modules

API Key

The API Key interface allows for the creation of API keys for Bright Server.

Method: create

HTTP Model:

- Verb: POST
- Form: (http|https):///api_key?param1=value1&...

Parameters

- sc*app*id (optional)
- sc*secret*key (optional)
- realm_guid (optional)

- realm*secret*key (optional)
- user_email (optional)

You must use either sc*app*id and sc*secret*key OR realm*guid and realm* secret_key.

Return Data

HTTP Codes

Response Body

Example:

```
curl -w "%{http_code}" -d 'sc_app_id=HN0CR2W8J8
&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N
&user_email=admin@aura-software.com' http://[BRIGHT URL]/bright/api/v2/api_key/

{
 "access_token":"a440b1ad868bc76716d22b4b827db77e",
 "created_at":"2012-12-18T12:16:29Z",
 "expires_at":null,
 "id":139,
 "course_provider_id":6,
 "token_type":null,
 "updated_at":"2012-12-18T12:16:29Z",
 "user_id":12
}
201
```

Note that this is a POST request, so you must configure your client appropriately. In the case of curl, this is accomplished with the **-d** flag.

The '-w %{http_code}' piece allows you to view the returned HTTP code.

Details

- If you do not set user_email, the generated token will have unlimited (administrative) access to the SCORMCloud application specified.
- If you specify an email that is new to the system, an account is automatically created for that user, and the new user is attached as a subscriber class user to the Bright Realm attached to the this SCORM Cloud application.

Errors

- If you do not specify a valid sc*app*id or sc*secret*key, you will receive HTTP error 401 (unauthorized).
- If you specify a user_email that is not valid, you will receive a 500 server error.

Scorm Cloud Registration

Method: index

The index method allows the api user to query the scorm*cloud* registrations.

Parameters

- **access method** - see Access Modes
- last_only (any value) - when included in the parameter string, we will return a single registration record, as demarcated by the last one created in SCORMCloud.
- sc*course*id: filter registrations to the specified course id.
- refresh*if*launched: if the last course launched is newer than the last crawl date, this will force the record to recrawl prior to returning a result.

Method: create

Creates a new scorm cloud registration. Or can be a single call that can be used to return an existing registration if one exists, or create a new one if it does not.

Parameters

- **access method** - see Access Modes
- **sc*course*id**: SCORMCloud course ID (required)
- **learner_id**: email address of user (required)
- **dont_duplicate**: check for an existing registration for this course and user, just return that if found. (optional)
- **check*scorm*cloud**: check the SCORMCloud API for existing registrations as well as the local Bright database

(optional). This can be useful in test scenarios where Bright SCORMCloud crawler is not functioning, and it is desirable not to create a lot of duplicate registrations.
- **fname** - Can be used to set the first name in the SCORMCloud registration.
- **lname** - Can be used to set the last name in the SCORMCloud registration.

Return Codes

- 201: a new record was created
- 302: a existing record was found (API call must set **dont_duplicate**)
- 400: bad request (generally a parameter is missing, such as sc*course*id).
- 500: server error. Please report this to aura support with the time of date received.

Return Data

For a successful request returned will be an XML or JSON document of the SCORMCloud registration:

```
{
 "attempts"=>2,
 "complete"=>"complete",
 "course_result"=>'<?xml version="1.0" encoding=...?> <rsp>...</rsp>',
 "crawl_error_msg"=>nil,
 "crawl_status"=>"success",
 "crawl_type"=>"course",
 "created_at"=>"2012-12-21T08:39:12Z",
 "full_result"=>nil,
 "id"=>5336,
 "last_crawled_at"=>"2012-12-12T16:33:33Z",
 "learner_id"=>"bret@aura-software.com",
 "number_of_crawl_errors"=>0,
 "sc_completed_at"=>nil,
 "sc_course_id"=>"20-NSFoundationPostTrainingQuiz",
 "sc_created_at"=>"2012-12-12T16:32:04Z",
 "sc_deleted"=>nil,
 "sc_err_code"=>nil,
 "sc_error_message"=>nil,
 "sc_last_accessed_at"=>nil,
 "sc_registration_id"=>"79c0dd35-139-048258-16e-9a1561b0a85d",
 "score"=>99.2,
 "course_provider_id"=>10,
 "course_id"=>509,
 "success"=>"succeeded",
 "totaltime"=>100.0,
 "updated_at"=>"2012-12-21T08:39:12Z"
}
```

In case of error, a message including the error is returned (error code 403):

```
{
"error_code":"4",
"error_message":"The maximum number of registrations has been reached.",
"error_url":"http://cloud.scorm.com/api?email=..."
}
```

Please note if you are using JSONP, since there are limited facillities to capture errors via JSONP, we will return error code 200, with the error block above. Otherwise you might find it difficult to correctly handle this error.

Method: gcreate

Since JSONP only works with "GET" requests, we provice a "GET" version of the create method called 'gcreate'. All else is identical to the 'create' method above, although the response will be formatted with the correct callback information so it can be processed via JSONP.

Method: launch_url

Returns URL for launching a course referencing by a given registration on scormcloud. This is a show action and should be called the same way, like:

```
http://bright.example.com/bright/api/v2/registration/97cd0d53-3190-4285-81e6-a916510ba58d/launch_url
```

Parameters

- **access method** - see Access Modes
- The id field is the sc*registration*id. It should look similar to the example.
- **redirect_url**: A URL to redirect to when the course is exited (required)
- **launching**: setting this to non null indicates your intention is to launch the course. We therefore set the "launched*at" record on the record to **now**. Later on, this value is used to determine if we need to force a recrawl of the registration record. See the index method and the 'refresh*if_launched' parameter.
- **tags**: A comma separated list of tags to apply to the online course. This maintains backward compatibility with the original Rustici SCORMCloud plugin, where a course was magically tagged with the post categories. If tags are set here, they will be tagged as such in SCORMCloud (if the course is a SCORMCloud course).

Scorm Cloud Course

Method: Index

Debugging Errors

The Bright API, if it's not happy with its input, isn't going to give you that much information as to why. It doesn't produce error messages other than HTTP codes. This is primarily for security reasons. Since we listen on public ports, we don't want people probing the api, and using the error messages as a guide to on how to gain access.

From curl, if you put an invalid response, you'll get no data back:

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?sc_app_id=HN0CR2W8J8
&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N'
```

We asked for a jsonc format, which of course doesn't exist.

The only information you will get is an HTTP error code, in this case 406, 'Not Acceptable'.

Fetching HTTP Error Codes from a GET Request

You can see the HTTP response code for a GET request in curl with the -I flag:

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?
sc_app_id=HN0CR2W8J8&
sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N' -I
 HTTP/1.1 406 Not Acceptable
 Content-Type: text/html; charset=utf-8
 X-Ua-Compatible: IE=Edge,chrome=1
 Cache-Control: no-cache
 X-Request-Id: d4da4e1010b84640e1657b731ada79a3
 X-Runtime: 0.004287
 Date: Fri, 07 Dec 2012 23:04:10 GMT
 X-Rack-Cache: miss
 Content-Length: 0
 Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-04-20)
 Connection: Keep-Alive
```

You will also get no data from a request that matches no data. But in this case, the HTTP code will be 200

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?sc_app_id=HN0CR2W8J8\
&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N&title=nosuchcourse'
```

[]

```
curl 'http://[BRIGHT URL]/bright/api/v2/course.json?sc_app_id=HN0CR2W8J8
&sc_secret_key=nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N&title=nosuchcourse' -I
  HTTP/1.1 200 OK
  Content-Type: application/json; charset=utf-8
  X-Ua-Compatible: IE=Edge,chrome=1
  Etag: "d751713988987e9331980363e24189ce"
  Cache-Control: max-age=0, private, must-revalidate
  X-Request-Id: 73a4966a2a62ca4e70316f6a68645b51
  X-Runtime: 0.006220
  Date: Fri, 07 Dec 2012 23:06:40 GMT
  X-Rack-Cache: miss
```

```
  Content-Length: 0
  Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-04-20)
  Connection: Keep-Alive
```

Fetching HTTP Error Codes from a POST Request

Since we use a RESTful API, some operations require a POST HTTP verb. For whatever reason, the -I flag in curl will not show you the HTTP return code for a POST.

Instead use the following:

```
curl -w "%{http_code}"  -d 'sc_app_id=HN0CR2W8J8x&sc_secret_key=\
nCwrTDSy1MzaeyhN0TFfi3uH3huzlu6CNmyHUG5N&user_email=newuser@aura-software.com' \
http://[BRIGHT URL]/bright/api/v2/api_key
```

-w %{http_code} is the key part of that.

Debugging Authentication Errors

If you receive a 401 code from the API, the server itself has recorded the authentication error in the server log. If you are completely stumped, this is the place to look.

An error like this should exists:

```
Processing by ScormCloudCourseController#index as JSON
  Parameters: {"api_key"=>"bogus!", "sc_course_id"=>"course1"}
   [1m [36mApiKey Load (0.4ms) [0m   [1mSELECT "api_keys".* FROM
  "api_keys" WHERE "api_keys"."access_token" = 'bogus!' LIMIT 1 [0m
Unauthorized: api token not found
Filter chain halted as :restrict_access rendered or redirected
Completed 401 Unauthorized in 2ms (ActiveRecord: 0.4ms)
```