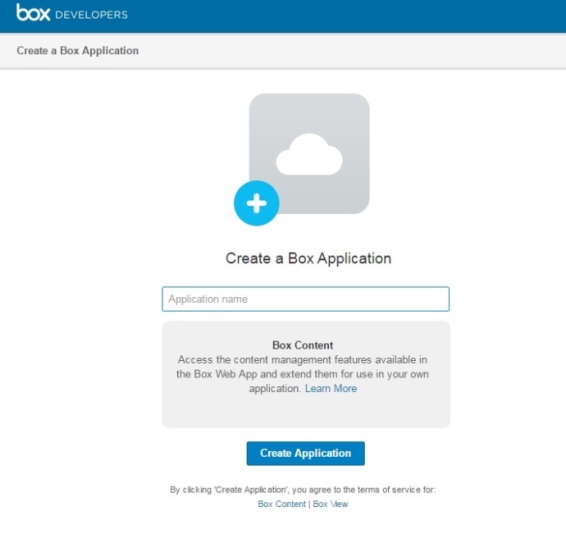
Created by Medair (Arnaud Mader), 29.06.2016

# How to integrate Box into Wigii by using an Enterprise Box App

The goal is to describe how to create an enterprise Box’s app, how to use it from a Wigii web site, how to obtain the different tokens and how use them.

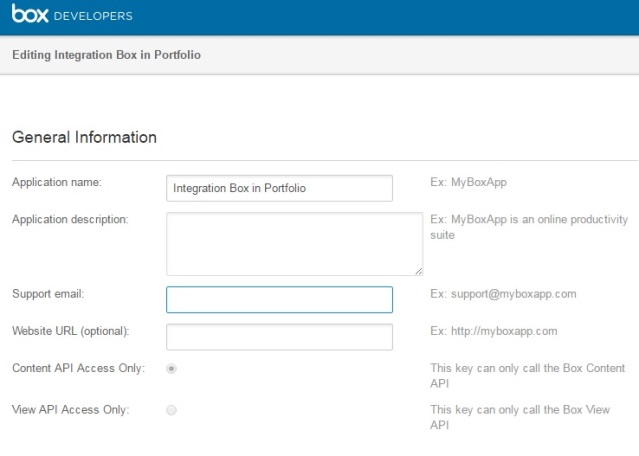
The Box’s app will access Box on behalf of the users, but we need to have each user authenticate with Box to both verify their identity and grant the application permission to access the content stored in their Box account.

## Create the Box’s app



On the box’s developer site, click on “Create a Box Application”.

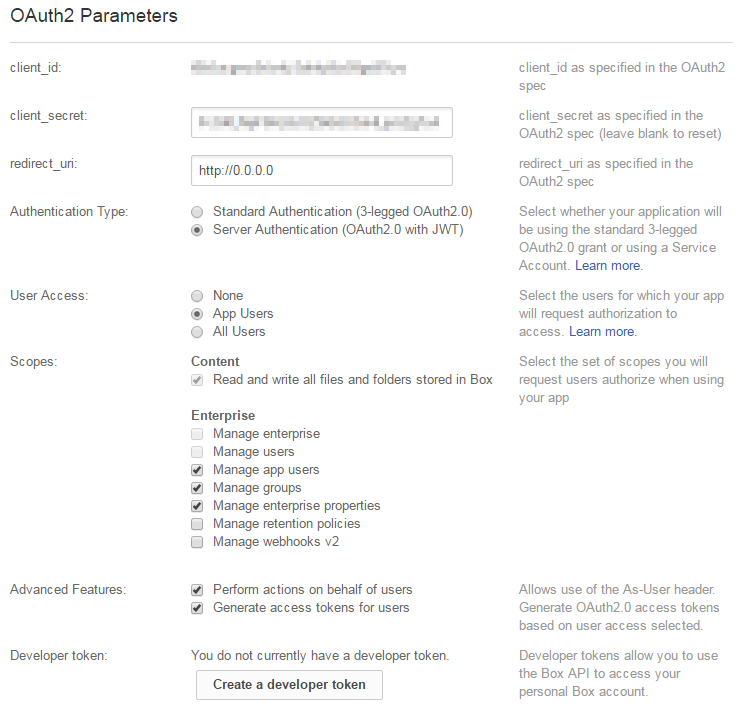
Then we must to choose an application’s name and click on *Create Application*.



Once we have created the application, we must to parameterize the app.

It is possible to add an application’s description or a support email for example.

**OAuth 2.0** is a protocol that allows the app to request a user’s authorization to access content in his or her Box account.

The *client\_id* and *client\_secret* are used for authentication and authorization purposes in the OAuth2 flow. It is very important to keep the both information secure.

To manually get the *refreshToken* and *accessToken* (see chapter 3 *Get Token*) using Postman, set the *redirect URL*, to <http://0.0.0.0>

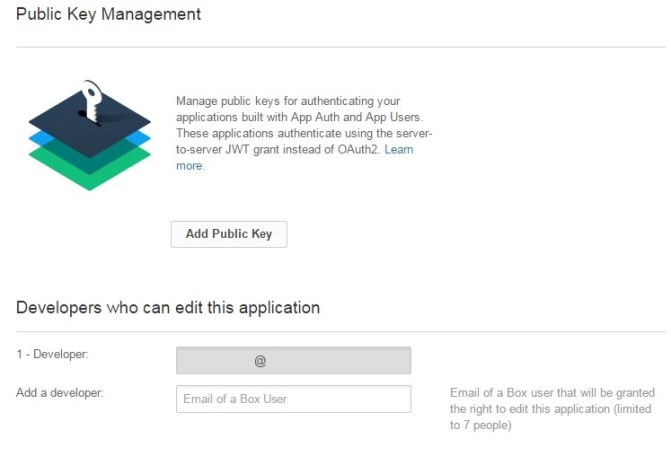
The authentication type should be set to *Server Authentication (OAuth2.0 with JWT)*

An ***App******User*** is a Box account that belongs to the Box Platform application, not an end-user of Box. Unlike typical Box accounts, these accounts do not have associated login and can only be accessed through the Box API.

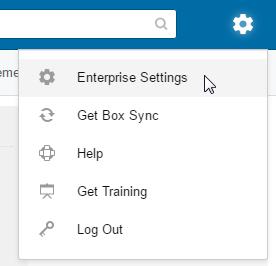
We must select the “*Read and write all files and folders*” scope if the application needs to perform operations on files and folders, such as creating, downloading, editing and deleting files and folders.

The enterprise’s scopes permit to define some permission.

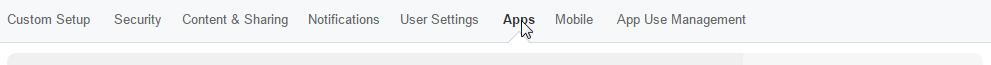
|  |  |
| --- | --- |
| **Read and write all files and folders** | Allows the application to upload, view, download file versions, and update the current file version.  Allows the application, edit and delete collaborations, tasks, comments, @mentions, task assignments, notifications, and collections. Allows the application to view the enterprise profile. |
| **Manage an enterprise** | Full set of scopes available for performing enterprise management, which includes permissions to view, create, edit, and delete users, content, collaborations, groups, reports, and admin settings. |
| **Manage an enterprise’s managed users** | Subset of the **Manage an enterprise** scope  Allows the application to add, view, edit, delete, activate, and disable standard Box users.  Allows the application to change the primary login, reset password, and change role for managed users, as well as manage enterprise content. |
| **Manage an enterprise's groups** | Subset of the **Manage an enterprise** scope  Allows the application view, create, edit, and delete groups and group memberships. |
| **Manage an enterprise's properties** | Subset of the **Manage an enterprise** scope  Allows the application to view and edit enterprise attributes and reports; edit and delete device pinners. |
| **Manage an enterprise's retention policies** | Allows the application to view and create content retention policies with Box Governance. |
| **Create and manage app users** | Allows the application to provision and manage its own App Users using the App Auth feature |



We can define the developers who can edit the application by adding their mails.



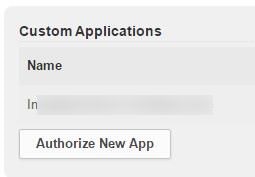
In order to begin creating App Users in an enterprise, an authorization for the application must be granted in the **Enterprise Admin** console.

In the **Enterprise Settings** we must click on **Apps**.

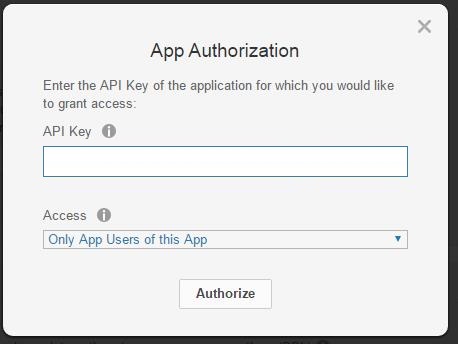
There are two places where we must to add the API Key (or client\_id).



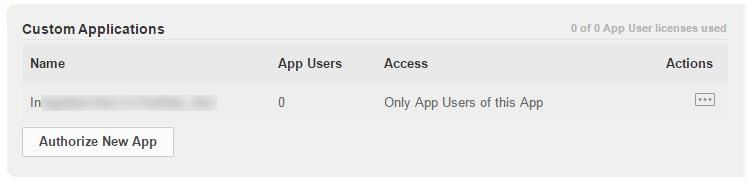
The first one is the ***Application Settings***, where we must to add the API Key in the field “*Except for*”:



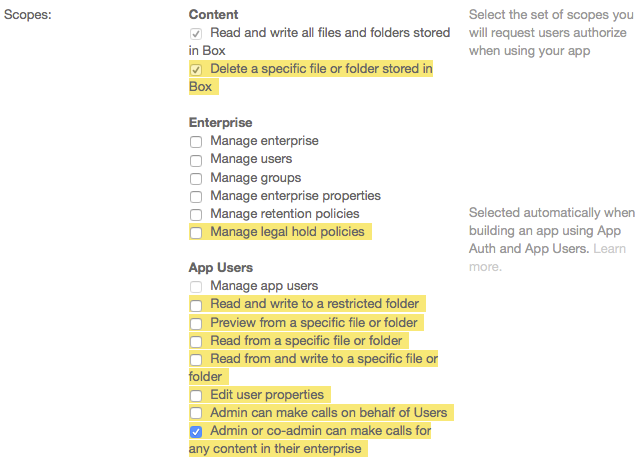
The second one is the ***Custom Applications***section; choose “*Authorize New App”*:



Then enter the API Key for the app we would like to enable and select an access level. This determines which users the app has control over.

After adding the API Key for all the applications, the Custom Applications part must list all the authorized applications that are active:

For authorize the application to access all folders in the Enterprise box’s account, we must have the following parameterization:

The App should be of type ***Standard Box App*** (not *App User*) with at least the default permission ***Read and write all files and folders stored in Box***

Ask the Box support to activate the options ***Admin and co-admin can make call for any content in their enterprise*** and ***Admin can make calls on behalf of Users (authorize to pass As-User header)***

But once the Box support activates the option, we need to go through the OAuth login process again to ensure that the new scopes take effect on our application (see the next chapter *Get Access Code*).

## Get access code

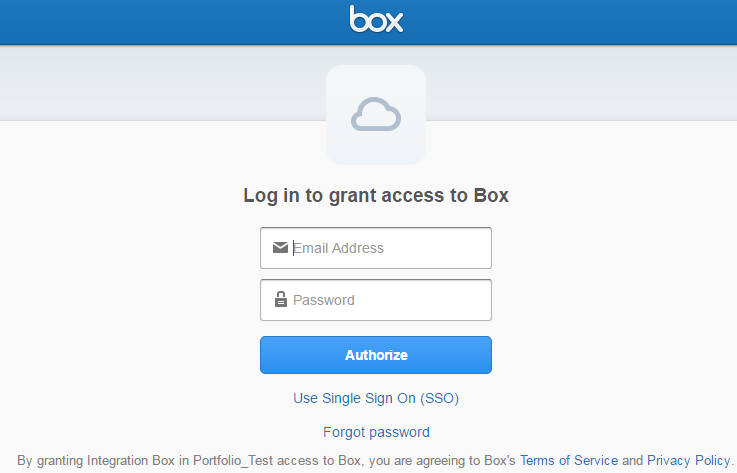
The first step is to obtain the authorization code:

We must to go to the address ***https://account.box.com/api/oauth2/authorize*** with a POST or a GET request with the parameters:

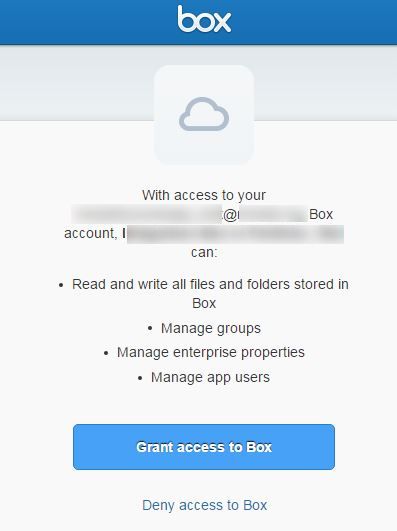
|  |  |
| --- | --- |
| **response\_type** *(required)* | Whether the endpoint returns an authorization code. |
| **client\_id**  *(required)* | The **client\_id** from the apps that we have obtained in the Initial Setup |
| **redirect\_uri**  *(required)* | An HTTPS URI or custom URL scheme where the response will be redirected. Must be registered with Box in the application console. |
| **state**  *(required)* | An arbitrary string that we could choose that will be included in the response of the applications. |
| **box\_login** | The email address of the user that we want to have pre-populated in the login form. |

We can use the following link (by replacing *MY\_CLIENT\_ID* by the app’s client\_id):

https://account.box.com/api/oauth2/authorize?response\_type=code&client\_id=MY\_CLIENT\_ID&state=authenticated



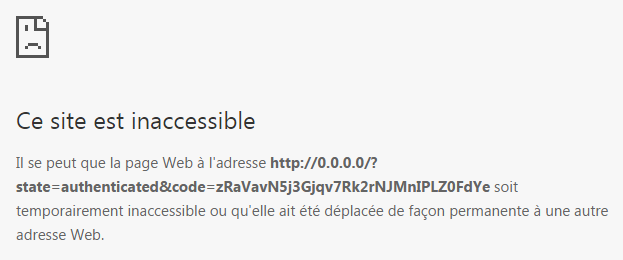
The link forwards us to the Box’s authorization page:



By using one of the developer’s Email address and the corresponding password we arrive to a consent page to authorize the application to access the account:

After clicking either **Grant** or Deny, we will receive a response from Box.

By clicking Grant we obtained the authorization code:



or the URL:



Ensure that when you copy-past an element or code to Postman that no spaces are at the end of the element or code. Postman takes each space as a character then an error will occur when you send the cUrl request.

## Get tokens

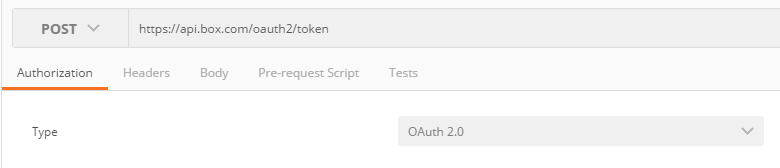
Once the application has completed and gotten an authorization code, it’ll now need to exchange the authorization code for an access token from Box, but quickly because the authorization code is only valid for **30 seconds**. If the authorization code is no more valid, re-do the last step.

To get the ***access\_token***, we’ll need to make a POST request to ***https://api.box.com/oauth2/token*** with the following parameters:

|  |  |
| --- | --- |
| **grant\_type** *(required)* | Must be **authorization\_code**. |
| **code** *(required)* | The authorization code we retrieved previously |
| **client\_id** *(required)* | *client\_id* from box Initial Setup. |
| **client\_secret** *(required)* | *client\_secret* from box Initial Setup. |

We could obtain all the tokens with the Postman’s application.

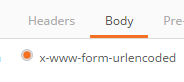
We choose POST (1), write the URL (2) and choose authorization **OAuth 2.0** (3).



**2**

**1**

**3**



We click on the Body and choose **x-www-form-urlencoded:**

And write the needed information (*grant\_type*, *client\_id*, *client\_secret* and *code*). After clicking on send we obtain the *access\_token* and *refresh\_token*.

The first one (*access\_token*) is used to access the Box application and is valid for **1 hour**.

The second one (*refresh\_token*) is used to refresh the access token when it is not ever valid, is validity is **60 days**.

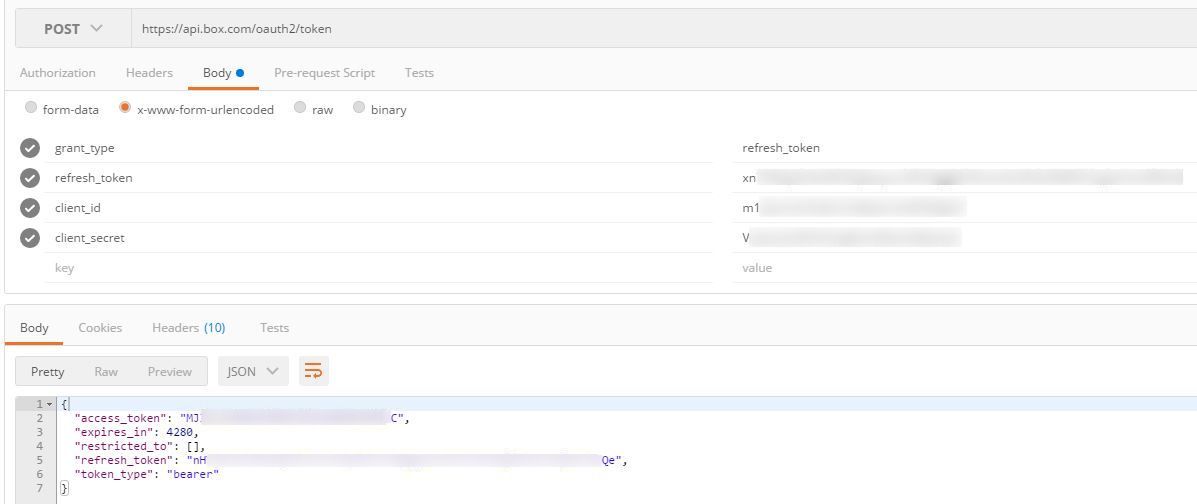
Every time we get a new *access\_token* by using a *refresh\_token*, we reset the timer for the 60 day period and hand us a new *refresh\_token*.

To use the *refresh\_token* to get a new *access\_token*, we must make a POST request to ***https://api.box.com/oauth2/token*** with the parameters:

|  |  |
| --- | --- |
| **grant\_type** *(required)* | Must be **refresh\_token**. |
| **refresh\_token** *(required)* | The currently active *refresh\_token* |
| **client\_id** *(required)* | *client\_id* from box Initial Setup. |
| **client\_secret** *(required)* | *client\_secret* from box Initial Setup. |

We can also use Postman if we want to refresh the *access\_token*.

We choose POST (1), writes the URL (2) and in the Body, we choose x-www-form-urlencoded (3) and write the parameters (*grant\_type*, *refresh\_token*, *client\_id*, *client\_secret*). After clicking send we obtain the new tokens’ pair.



**1**

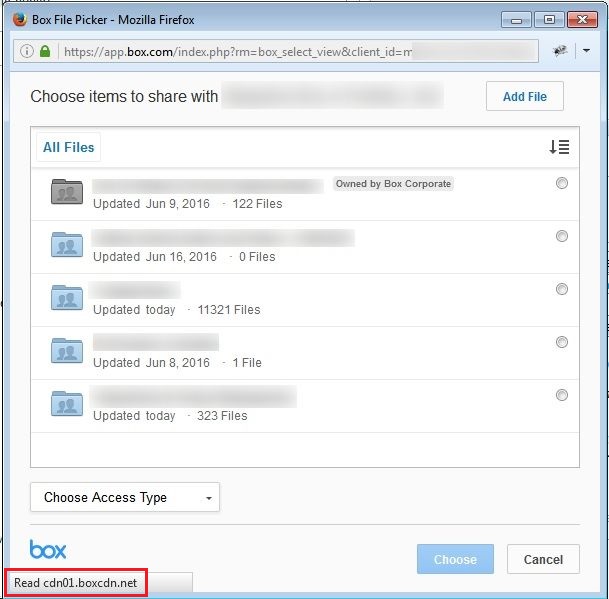
**2**

**3**

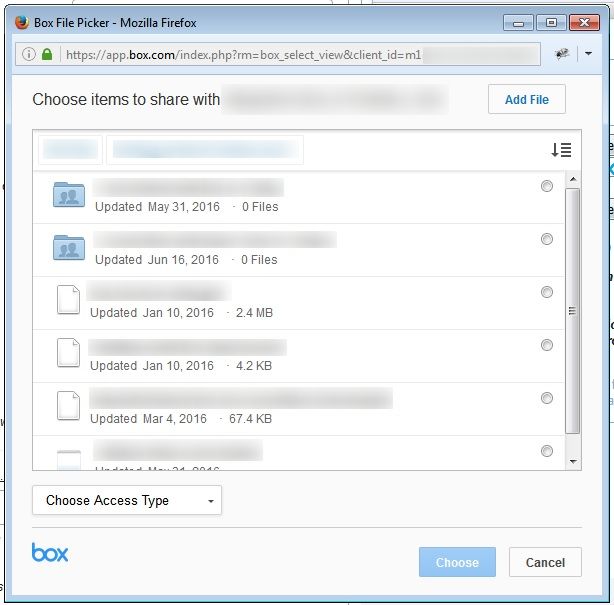
If the refresh token is no more valid (time’s up or loose it), we must to restart with the authorization code (chapter 2).

## Use the Box’s File Picker

The File Picker is configured that only one file can be added.



Once we click on the Box button to add a file stored on Box, we must waiting until the message “*Read cdn01.boxcdn.net*” **disappears**.



**2**

To add a file we must to click on the radio button (1) and finally on “Choose”.   
Folders (2) **can’t** be chosen to be uploaded as document.

**1**

The “Add File” button opens the users’ Box account on the default browser.

## How to use it in the code

### File Picker

We can choose a file by using the File Picker, it allows the user to share their Box content with the application. The user needs to introduce his Box’s username and password.

In developer side, there are two ways to use the file picker, with HTML Widget or with JavaScript APIs. The second way allows adding some additional control and capabilities.

#### Using the HTML Widget

With the HTML Widget, we can have a file picker with 3 steps:

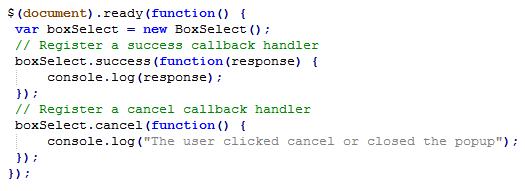
##### Include Box’s JavaScript Library:



##### Configure the Box File Picker



##### Subscribe to User Actions (JavaScript)



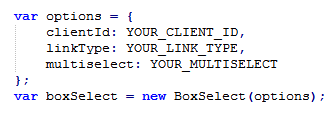
#### JavaScript APIs

Like the HTML Widget, the JavaScript APIs require to include Box’s JavaScript library, configure Box file picker, and subscribe to user actions.

##### Include Box’s JavaScript Library (HTML)



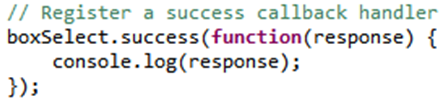
##### Configure the Box File Picker (JavaScript)



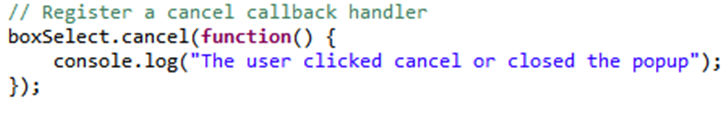
##### Subscribe to User Actions (JavaScript)

There are two possible users’ actions

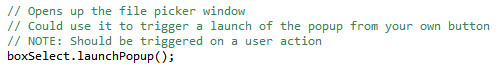
* + - 1. **Success** when a user successfully selects a file or a folder, a response object is passed to the success call-back handler

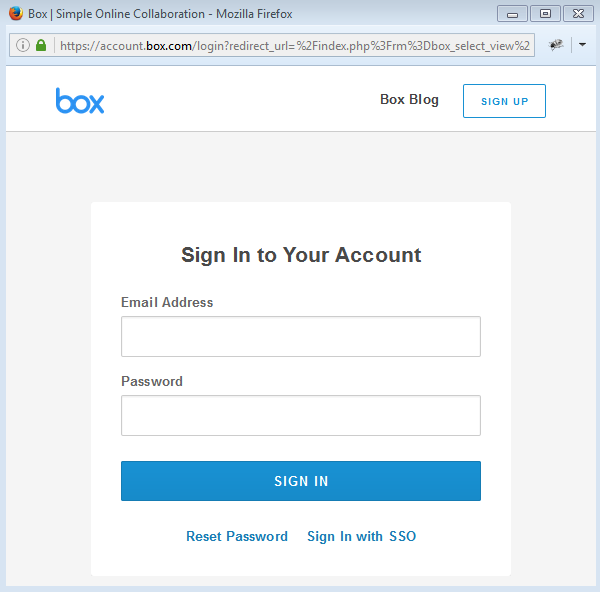


* + - 1. **Cancel** when a user clicks cancel in the Box file picker or close the Box file picker popup window, the cancel call-back handler will be called.



##### Launch the File Picker Popup





That will open a window like:

To access the files we need to put our Box’s email address and password.

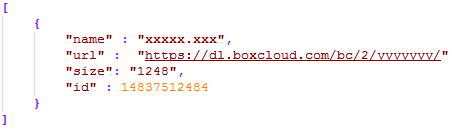
#### Configuration Options

|  |  |
| --- | --- |
| **client\_id** | The *client\_id* or the API key of the application. |
| **link\_type** | The type of the link can be *direct* or *shared*. The direct link is only valid for 15 minutes. |
| **multiselect** | Allows the user can select a single item or multiple items, can be *true* or *false*. |

#### Response

###### Direct Link

If we configure the link type to be a direct link, the response will be a JSON snippet and look like this:



|  |  |
| --- | --- |
| **name** | The temporary URL of the file |
| **url** | Name of the item |
| **size** | Size of the file in bytes |
| **id** | The ID of the file. This ID can be used only to uniquely identify a file in the Box Content API. |

*Remember that the URL is valid only* ***15 minutes****.*

###### Shared Link

If we configure the link type to be a shared link, the response will be a JSON snippet and look like this:



|  |  |
| --- | --- |
| **url** | The URL of the file |
| **name** | Name of the item |
| **access** | The group that can access the link can be *collaborators*, *company*, and *open*. |
| **type** | The type of the item can be *file* or *folder*. |
| **id** | The ID of the file. This ID can be used only to uniquely identify a file in the Box Content API. |

### Box Config File

We use a XML’s file for stock all information concerning the Box’s access, the file is:

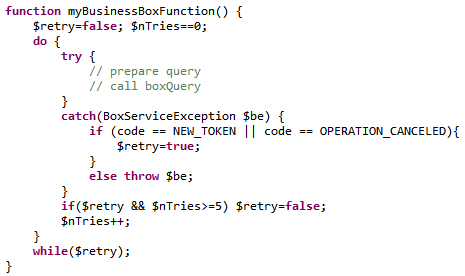
Where

* *boxWebUrl* is the URL for open the root site of the Box account.
* *refreshUrl* is the URL we must call to refresh the both tokens.
* *boxUrl* is the URL we must call to have information, but some additional information must be added in function what we want have.
* *boxUpload* is the URL for the upload of a box’s file and for the update version of a box’s file
* *boxFolderUrl* is the URL for information about a box’s folder.
* *modificationDate* is the date where the *accessToken* and *refreshToken* have been modified the last time.
* *userId* is the id of the user that creates the application (see Get Access Code part).
* *statusDate* is the date where the *lock* has been set the last time to Idle.
* *lock* is the lock to avoid two people to access the file at the same time. *Idle* is when it is possible to access the document and *Busy* when it is not possible to access it.

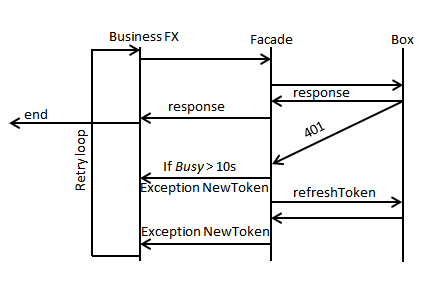
### BoxServiceFormExecutor pattern

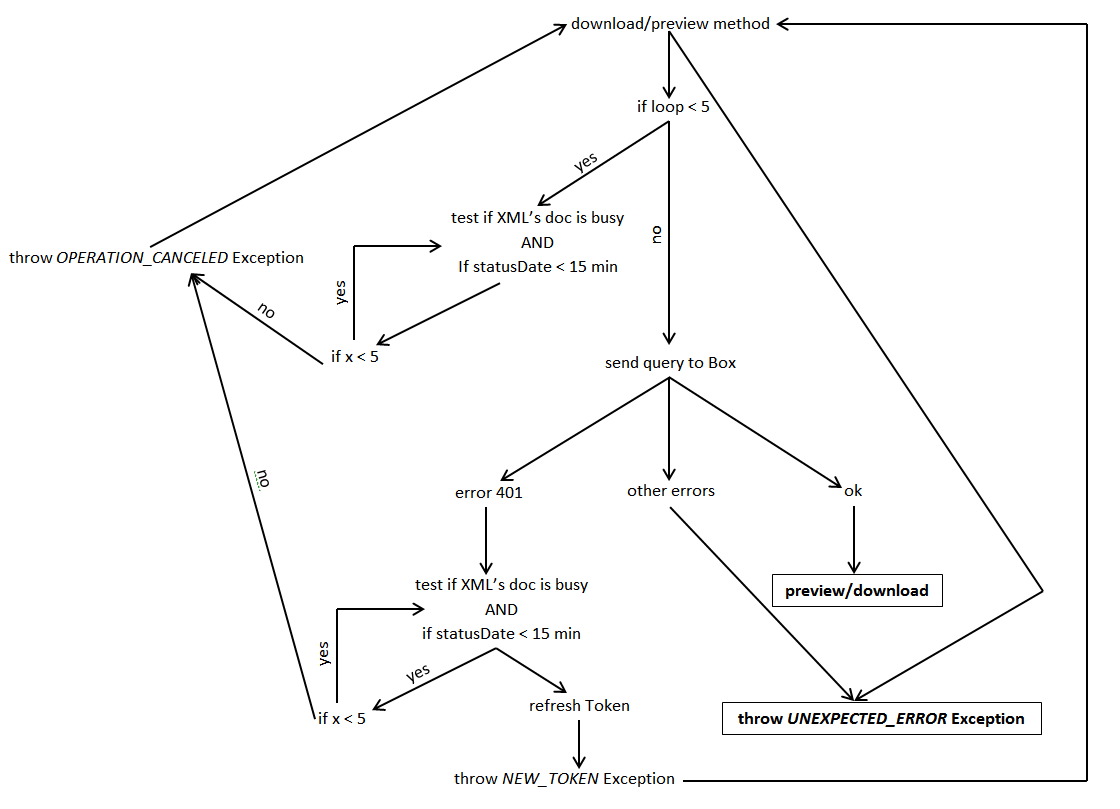
For doing the download and the preview of the files’ Box, we have created a new service, the *BoxServiceFormExecutor*, that permits to keep the old behaviour and "short circuit" the normal behaviour in the case of Box and call the new service.

The pattern to obtain the different information from Box is:



And the call to boxQuery is:

The *401 Error* is the Box’s response if the *accessToken* is no more available and when we must to refresh it. Otherwise we have the file’s URL (embed link to preview or download’s link) and then can continue the execution of the code.

When we receive the error response from Box we check if the XML’s document is busy or not to modify the *accessToken*, if yes we wait 10 seconds and retry to execute the query. We do that 4 times, if after the 4th tries we can’t refresh the tokens, we stop the execution and throw an Exception.

It is necessary to throw a *NewToken* exception because the query header contains the *accessToken* and if we modify it, we must rebuild the query with the new *accessToken*.

If there are no errors we end the execution with the download or the preview, but for all error, we exit the execution with an *UNEXPECTED\_ERROR* Exception.

### Get File information

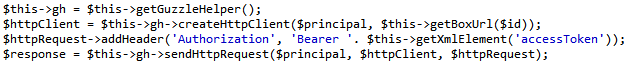
In the code it is possible to have more information about the file, for that we need to have the file’s ID (see previous section for know how to obtain it).

The cURL request is:

curl https://api.box.com/2.0/files/FILE\_ID -H "Authorization:

Bearer ACCESS\_TOKEN"

we use Guzzle to initiate HTTP request to the server. Guzzle is PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services.



Where

* + - * *getBoxUrl()* returns ***https://api.box.com/2.0/files/ID\_FILE***
      * *getXmlElement('accessToken')* returns the Access Token stored in the XML document.

The variable *$response* contains all the JSON like:

{

"type": "file",

"id": "5000948880",

"file\_version": {

"type": "file\_version",

"id": "26261748416",

"sha1": "134b65991ed521fcfe4724b7d814ab8ded5185dc"

},

"sequence\_id": "3",

"etag": "3",

"sha1": "134b65991ed521fcfe4724b7d814ab8ded5185dc",

"name": "nameOfFile.xxx",

"description": "description of the file",

"size": 629644,

"path\_collection": {

"total\_count": 2,

"entries": [

{

"type": "folder",

"id": "0",

"sequence\_id": null,

"etag": null,

"name": "All Files"

},

{

"type": "folder",

"id": "11446498",

"sequence\_id": "1",

"etag": "1",

"name": "Pictures"

}

]

},

[…]

"parent": {

"type": "folder",

"id": "11446498",

"sequence\_id": "1",

"etag": "1",

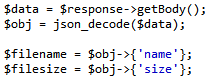
"name": "Pictures"

},

"item\_status": "active"

}

All the information can be accessed with the following code for example:





*in the example will return 0*



*in the example will return 11446498*

The PHP method *json\_decode()* takes a JSON encoded string and converts it into a PHP variable, this function only works with UTF-8 encoded strings.

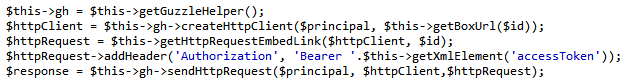
### Get temporary URL

Is quite similar of the previous section but we need to add additional information (parameter *fields)*.

curl https://api.box.com/2.0/files/FILE\_ID?fields=expiring\_embed\_link \

-H "Authorization: Bearer ACCESS\_TOKEN"

The new parameter is added in the function *getHttpRequestEmbedLink()*.



The variable *$response* contains the response with the “*expiring\_embed\_link*” field if the ID is valid and if the user has access to the file. This field is a JSON object with another filed called “*url*” in which we will find the link to use in an embedded iframe to provide a rendering of the file.

{

"type": "file",

"id": "123456",

"etag": "1",

"expiring\_embed\_link": {

"url": "https://app.box.com/preview/expiring\_embed/gvoct6FE!Qz2rDeyxCiHsYpvlnR7JJ0SCfFM2M4YiX9cIwrSo4LOYQgxyP3rzoYuMmXg96mTAidqjPuRH7HFXMWgXEEm5LTi1EDlfBocS-iRfHpc5ZeYrAZpA5B8C0Obzkr4bUoF6wGq8BZ1noN\_txyZUU1nLDNuL\_u0rsImWhPAZlvgt7662F9lZSQ8nw6zKaRWGyqmj06PnxewCx0EQD3padm6VYkfHE2N20gb5rw1D0a7aaRJZzEijb2ICLItqfMlZ5vBe7zGdEn3agDzZP7JlID3FYdPTITsegB10gKLgSp\_AJJ9QAfDv8mzi0bGv1ZmAU1FoVLpGC0XI0UKy3N795rZBtjLlTNcuxapbHkUCoKcgdfmHEn5NRQ3tmw7hiBfnX8o-Au34ttW9ntPspdAQHL6xPzQC4OutWZDozsA5P9sGlI-sC3VC2-WXsbXSedemubVd5vWzpVZtKRlb0gpuXsnDPXnMxSH7\_jT4KSLhC8b5kEMPNo33FjEJl5pwS\_o\_6K0awUdRpEQIxM9CC3pBUZK5ooAc5X5zxo\_2FBr1xq1p\_kSbt4TVnNeohiLIu38TQysSb7CMR7JRhDDZhMMwAUc0wdSszELgL053lJlPeoiaLA49rAGP\_B3BVuwFAFEl696w7UMx5NKu1mA0IOn9pDebzbhTl5HuUvBAHROc1Ocjb28Svyotik1IkPIw\_1R33ZyAMvEFyzIygqBj8WedQeSK38iXvF2UXvkAf9kevOdnpwsKYiJtcxeJhFm7LUVKDTufuzuGRw-T7cPtbg.."

}

}

And we obtain the URL through:



The URL will expire after 60 seconds and the preview session will expire after 60 minutes.

### Download a file

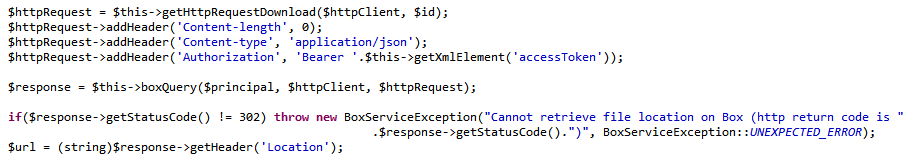
If the file is available to be downloaded, the response will be a *302 found* to a URL at *dl.boxcloud.com* (which is not persistent). In otherwise if the file is not available to be downloaded, a response with an HTTP status *202 accepted* will be returned with a *Retry-After* header indicating the time in seconds after which the file will be available for the client to be downloaded.

The cUrl request is:

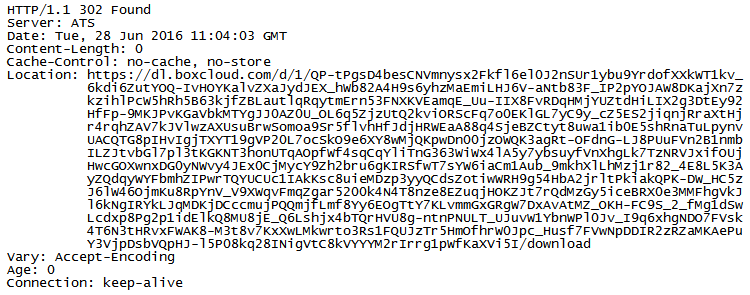
curl -L https://api.box.com/2.0/files/FILE\_ID/content

-H "Authorization: Bearer ACCESS\_TOKEN"

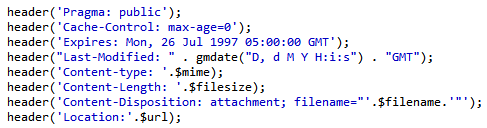
Once we have recovered the file’s information, we initiate the cUrl exchange:



The response header contains this information:



If the file is available to download (code 302), we recover the URL in the Location and initiate the download:



### Refresh the Tokens

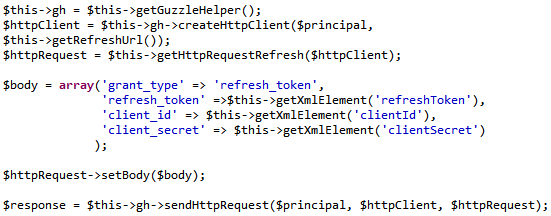
The cUrl is of the form:

curl https://api.box.com/oauth2/token \

-d 'grant\_type=authorization\_code&code=YOUR\_AUTH\_CODE&client\_id=YOUR\_CLIENT\_ID&client\_secret=YOUR\_CLIENT\_SECRET' \

-X POST

If the access token is no more valid, we must to refresh the tokens. To do that, we do:



where *getRefreshUrl()* returns ***https://api.box.com/oauth2/token***

*$body* contains all the needed information to refresh the tokens, stored in the XML document

The *$response* contains the new *access\_token* and *refresh\_token*, in the form:

{

"access\_token": "T9cE5asGnuyYCCqIZFoWjFHvNbvVqHjl",

"expires\_in": 3600,

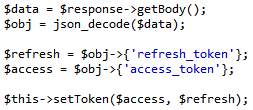
"restricted\_to": [],

"token\_type": "bearer",

"refresh\_token": "J7rxTiWOHMoSC1isKZKBZWizoRXjkQzig5C6jFgCVJ9bUnsUfGMinKBDLZWP9BgR"

}

We now need to store this information in the document:



where the *setToken()*’s method modify the XML document to update the both fields *access\_token* and *refresh\_token*.

### Upload and Update a file

Two tags can be added on Element fields of type Files:

***boxAllowUpdate*** with values 0 or 1

***boxFolderId*** with value a box’s folder Id.

The behaviour of the code depends of these two tags.

###### boxAllowUpdate = 0 and boxFolderId is not given

The behaviour is the standard one.

If we choose a local’s file, on save, the file is transferred on the server, through a temporary file.

If we choose a box’s file, on save, the link “box://fileId” is added on the database in the path field of the file’s.

###### boxAllowUpdate = 0 and boxFolderId is given

If we choose a local’s file, on save, the file is uploaded in box in the folder defined by the tag “boxFolderId” of the element, then a new file is added on the box’s folder and the link “box://fileId” is added on the database in the path field of the file’s table with the id of the new file. If a file with the same name and same type already exist in the folder, the version is updated with the new file and the link “box://fileId” is updated on the database in the path field of the file’s table with the id of the new file.

If we choose a box’s file, on save, the link “box://fileId” is updated on the database in the path field of the file’s table with the id of the new file.

###### boxAllowUpdate = 1 and boxFolderId is not given

If we choose a local’s file, on save, the file is transferred on the server, through a temporary file. Except is we replace a Box’s file in this case we update the old file.

If we choose a box’s file, on save, the link “box://fileId” is updated on the database in the path field of the file’s table with the id of the new file.

###### boxAllowUpdate = 1 and boxFolderId is given

If we choose a local’s file, on save, the file is uploaded in box or the version is updated if a box’s file already exist and if the mime type is the same.

If we choose a box’s file, on save, the link “box://fileId” is updated on the database in the path field of the file’s table with the id of the new file.

See the *Table box integration behaviour* 1(Annexe B) to have a summary of the configuration truth table.

The upload and update is only **for local’s file**, and consists of two distinct parts in the code.

The **upload** consists of adding the file in the given box folder. The cUrl’s request is:

curl https://upload.box.com/api/2.0/files/content \

-H "Authorization: Bearer ACCESS\_TOKEN" -X POST \

-F attributes='{"name":"fileName.xxx", "parent":{"id":"11446498"}}' \

-F file=@myfile.jpg

The *$response* contains a set of information (the following JSON is not complete):

{

"total\_count": 1,

"entries": [

{

"type": "file",

"id": "5000948880",

"sequence\_id": "3",

"etag": "3",

"sha1": "134b65991ed521fcfe4724b7d814ab8ded5185dc",

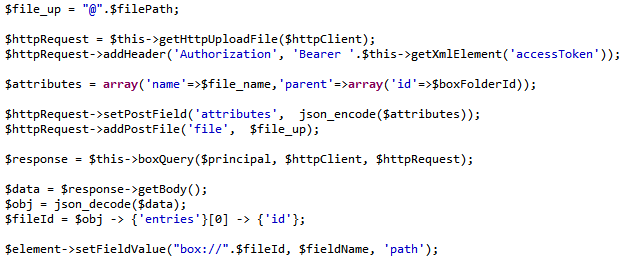
"name": "fileName.xxx",

"description": "description",

"size": 629644,

[…]

}

And the corresponding code is:

When we add a local file a temporary file in the temporary folder is created. We send this file to Box.

The variables:

* *getHttpUploadFile()* method return the Box’s URL for the file’s upload (see cUrl request).
* *$filePath* is the path to the temporary file in temporary folder.
* *$attributes* is an array that contains the name of the file and the parent folder’s id. We must to encode this array in JSON before adding this element in the post field of the request.
* *$boxFolderId* is the box’s folder id if given in the tag of the element.

Once we send the document to Box, we recover the new file’s id and set the path of the file with this id, to permit during the save to add the information *box://fileId* in the database.

The **update** consists of modifying an existing Box’s file with another one but of the same type. This update is only available if we have already a Box’s file id.

curl https://upload.box.com/api/2.0/files/FILE\_ID/content \

-H "Authorization: Bearer ACCESS\_TOKEN" \

-H "If-Match: ETAG\_OF\_ORIGINAL" \

-F filename=@FILE\_NAME

The *$response* contains a set of information (the following JSON is not complete):

{

"total\_count": 1,

"entries": [

{

"type": "file",

"id": "5000948880",

"sequence\_id": "3",

"etag": "3",

"sha1": "134b65991ed521fcfe4724b7d814ab8ded5185dc",

"name": "fileName.xxx",

"description": "description",

"size": 629644,

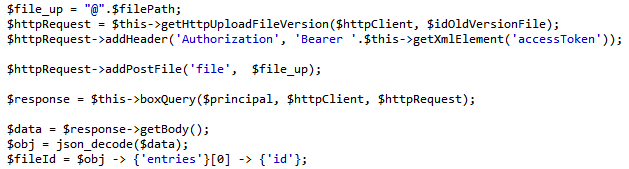
[…]

}

]

}

And the code:



The variables:

* *$filePath* is the path to the temporary file.
* *$idOldVersionFile* is the box’s file id of the existing file we want to update
* *getHttpUploadFileVersion()* method return the Box’s URL for the file’s update.

Once we send the document to Box, we recover the new file’s id and set the path of the file with this id, to permit during the save to add the information *box://fileId* in the database.

### File Metadata

Metadata allows users and applications to define and store custom data associated with their files. Metadata consists of a key: value pairs that belong to files (or folder). In Box it is possible to add metadata through Box’s API, there are two possible of metadata’s schema “Metadata Templates” and “Custom Metadata”. In our case we only use the “Custom Metadata”.

If there are no metadata on a file, we must to **create** this metadata with the following cUrl’s request (it is possible to add more key: value pairs):

curl https://api.box.com/2.0/files/ID\_FILE/metadata/SCOPE/TEMPLATE \

-H "Authorization: Bearer ACCESS\_TOKEN" \

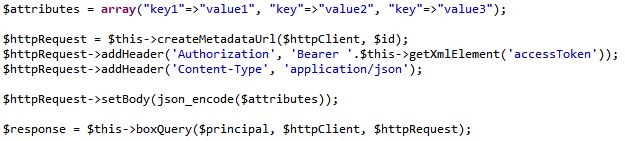
-H "Content-Type: application/json" \

-d '{"xxx": "value", "yyy": "value"}' \

-X POST

Where the *xxx* and *yyy* are the tags of the metadata and the both *value* are the corresponding value. It is possible to add more metadata.

In the code:



If a metadata already exists we must to use the following cUrl’s request to **update** the metadata :

curl https://api.box.com/2.0/files/ID\_FILE/metadata/SCOPE/TEMPLATE \

-H "Authorization: Bearer ACCESS\_TOKEN" \

-H "Content-Type: application/json-patch+json" \

-d '[{"op": "add", "path": "/xxx", "value": "no"},

{"op": "add", "path": "/yyy", "value": "no"}

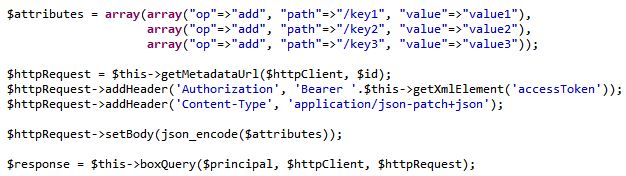
]' \

-X PUT

The “/” in the field path is needed to indicate that the metadata is located at the root.

In this example only the ***add*** operation is used, but other operations exist. The *add* operation allows to overwrite an existing value.

In the code:



The other possible operations are ***replace*** (replace an existing value by another), ***test*** (the value of the existing value must match the specified value), ***remove*** (delete a tag), ***move*** (move a value from another one) or ***copy*** (copy the value from another tag).

In the case of *add*, *replace*, and *test*, the line must be composed of an *op*, *path*, and *value*.

In the case of *move*, and *copy*, the line must be composed of an *op*, *from*, *path*, where the *from* represent the value to take and the *path* the element to modify.

In the case of *remove* the line must be composed of an *op*, and *path*.

As we are in a “Custom Metadata” schema, the **SCOPE** and **TEMPLATE** have respectively the value “***global***” and “***properties***”.

To have more information see: <https://docs.box.com/reference#update-metadata>

### Events & Long Polling

The Box’s event represents an event stream, a source of events that continues to make new events available as they occur, without limit.

To Begin working with the events endpoint we must first determine the current stream position to have the starting point. The following cUrl’s command retrieves the current stream position:

curl https://api.box.com/2.0/events?stream\_position=now \

-H "Authorization: Bearer ACCESS\_TOKEN"

The possible response is:

{

"chunk\_size": 0,

"next\_stream\_position": 1363125294286,

"entries": []

}

Where the “*next\_stream\_position*” represents the stream’s current position.

The next step is to get the URL to start the long poll. For that we can execute the following cUrl’s command:

curl https://api.box.com/2.0/events \

-H "Authorization: Bearer ACCESS\_TOKEN" \

-X OPTIONS

A possible response is:

{

"chunk\_size": 1,

"entries": [

{

"type": "realtime\_server",

"url": "http://2.realtime.services.box.net/subscribe?channel=e9de49a 73f0c93a872d7&stream\_type=all",

"ttl": "10",

"max\_retries": "10",

"retry\_timeout": 610

}

]

}

Where the “*url*” is the URL for the long poll.

Next, we must to begin to listen for events by sending a GET request to the URL. The “*stream\_position*” is optional but enables us to specify exactly where in the stream of event to begin listening for new ones.

curl https://api.box.com/2.0/events?stream\_position=1363125294286 \

-H "Authorization: Bearer ACCESS\_TOKEN"

A possible response (part of response):

{

"chunk\_size": 1,

"next\_stream\_position": 1363126800230,

"entries": [

{

"type": "event",

"event\_id": "1b40ca497d3f41f80f98c45bb4c7115fc3ba09da",

"created\_by": {

"type": "user",

"id": "10523870",

"name": "Ted Blosser",

"login": "ted+demo@box.com"

},

"created\_at": "2013-03-12T15:19:57-07:00",

"recorded\_at": "2013-03-12T15:19:57-07:00",

"event\_type": "ITEM\_PREVIEW",

"session\_id": "1902881648513faa0655d6a",

"source": {

"type": "file",

"id": "6857048632",

"sequence\_id": "1",

"etag": "1",

"sha1": "fa69067b83d526eb27d24fbb05eab0dfeee8a8f5",

"name": "nameOfFile.xxx",

"description": "",

"size": 1321661,

"path\_collection": {

"total\_count": 1,

"entries": [

{

"type": "folder",

"id": "1",

"sequence\_id": null,

"etag": null,

"name": "Trash"

}

]

},

[…]

}

}

]

}

The “*event\_type*” represents the type of event we receive, the others information we need is about the element that has started the event, the “*source*” give us these information.

Once we receive a response, the long poll ends. So we must restart it after retrieving the results. There are two possible responses, one with an event:

{

"message": "new\_change"

}

One without event:

{

"version": 1,

"message": "reconnect"

}

After both responses it is necessary to restart the long poll. After retrieving the results we simply need to repeat the steps so far. If no events have occurred since our first GET request to the long poll URL then we will receive a message to reconnect using another long-poll URL.

For more details see: <https://docs.box.com/v2.0/docs/using-long-polling-to-monitor-events>

### Webhook

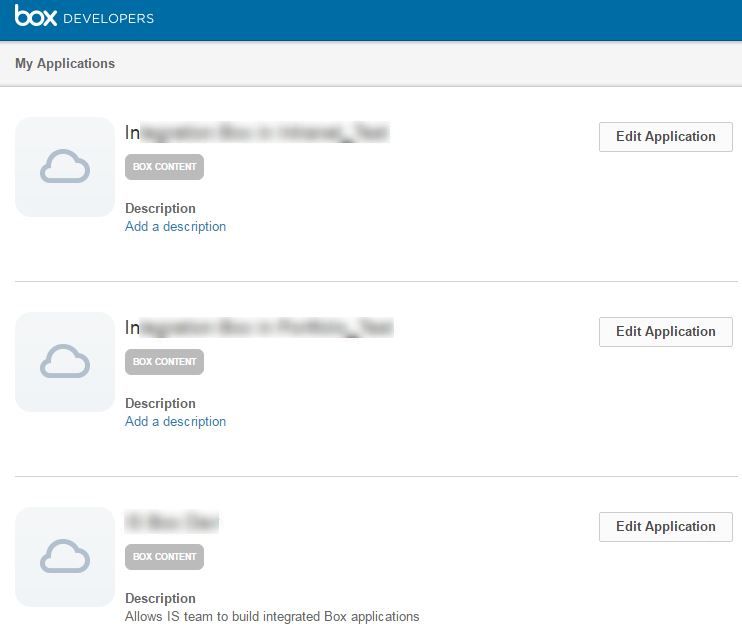
There are two versions of Webhooks which work differently.

A notification URL is needed, this URL must be a valid HTTPS URL. The URL must use the standard HTTPS port (443). The notification URL should be the endpoint of a service provided by our application for capturing and handling Webhook notifications. To signal that our service has accepted the notification, it should return an HTTP status in the range 200-299 within 30 seconds of receiving the notification.

#### Webhook v1

These Webhooks v1 are scoped to a user's whole account.

To activate this Webhook we must to follow the following steps:

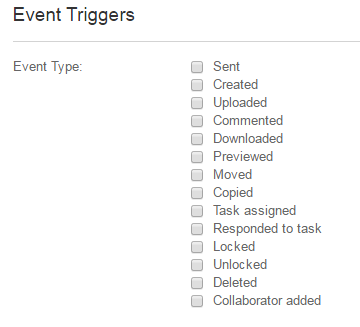


###### Go to the developer site (<https://developer.box.com/>) and log in. Once logged, we have the list of application that we have created.

###### Click on “*Edit Application*”, go down until the “*Webhooks*” part an click on “*Create a New Webhook*” to create new one or if some Webhooks already exists, we can click on “*Edit*” to modify an existing Webhook.

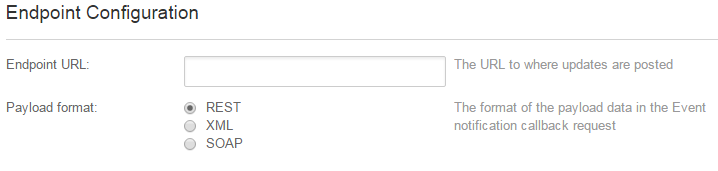
###### We are now in the creating part of the Webhook. We must to insert in the **General Information**section a name for the Webhook, a description and we can allow a user to disable the Event Listener.

###### The next section is the **Event Triggers**, in which we indicate which events the Webhook must monitor. When a monitored event occurs, Box sends a notification to the endpoint URL that we have specified (see next part).



###### The last part is the **Endpoint Configuration** in which we must to indicate an URL to which the Webhook must send a request (REST, XML or SOAP).

###### The URL that we supply must be a valid HTTPS URL that designates a web service that can accept and handle the notifications that Box sends.



###### Finally we can choose the information about monitored events that we want Box to send to the endpoint. The notification sends this information in **callback parameter**. Each callback parameter becomes a field in the notification data sent to the endpoint.

###### Once we have added all the call-back parameters we want to send to the endpoint, we must to click on “*Save Webhook*” button to create and activate the Webhook.

###### The Webhook will be triggered only if the application is used by a Box user.

###### We can see the Webhook on the following pages.

###### [*https://app.box.com/services/YOURAPPNAME*](https://app.box.com/services/YOURAPPNAME)

###### [*https://cloud.app.box.com/developers/services/edit/*](https://cloud.app.box.com/developers/services/edit/)

###### There are two ways to activate the Webhook inside the enterprise. First one is to ask each user within the enterprise to go through the OAuth2 process though and grant access to the Webhook. But if a user doesn't grant access to the Webhook, we will not get his/her events.

###### For the second one, we will have to publish the application to the Box App Gallery, by activate the “*Submit for approval*” option in the “*Information for listing in the Box App Gallery*” part of the Box’s app:



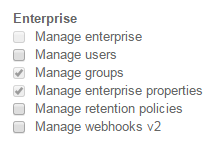
###### Then enable the application by default to our enterprise. Then un-publish the application. Un-publishing the application ensures that no other Box users that are not part of our enterprise will add the application.

For more information see: <https://docs.box.com/docs/webhooks>

#### Webhook v2

If we are looking to watch a specific folder or file, we can use the Webhooks v2. It notifies us when those objects are accessed. We have to set up the specific triggers we are looking to get back. When we set up the Webhook on a folder, the triggers will watch the all the related folders that are underneath it (it’s mean that if we have a folder called A, which we have set a webhook on. A has a child folder called B. Since A is the parent folder of B, any action that happens inside B will trigger the webhook set on A. This occurs because B is contained within A.)

###### First we must to activate the Webhook v2. In the application go to the **OAuth2 Parameter** and click on “*Manage Webhook v2*”:



###### Use the cUrl to create a new Webhook

curl https://api.box.com/2.0/webhooks \

-H "Authorization: Bearer <ACCESS\_TOKEN>" \

-H "Content-Type: application/json" -X POST \

-d '{"target": {"id": "<TARGET\_ID>", "type": "folder"}, "address": "<NOTIFICATION\_URL>", "triggers": [ARRAY\_TRIGGER]}'

Where we must to replace

* “*ACCESS\_TOKEN*” with our access token.
* “*TARGET\_ID*” with the ID number that Box assigns to the file or folder that we want to use as the target of our Webhook.
* “*NOTIFICATION\_URL*” with the HTTPS URL we want to send notifications to.
* “*ARRAY\_TRIGGER*” with the list of event we want to monitor (*FILE.UPLOAD*, *FILE.DOWNLOAD* …). For more trigger example see <https://docs.box.com/reference#event-triggers>

If we have a file instead of folder, we must to replace “folder” by “file” in the type.

###### To retrieve the Webhook, we can use the following cUrl

curl https://api.box.com/2.0/webhooks/4165 \

-H "Authorization: Bearer <ACCES\_TOKEN>" \

-H Cache-Control: no-cache

###### When Box observes an event that a Webhook is watching, it sends an HTTP request to the Webhook’s notification URL.

###### The body of a Webhook notification message is a JSON object that describes the Webhook’s target and the event trigger that sent the notification. The structure of the body is as follows:

{"id": A\_NOTIFICATION\_ID,

"type": "webhook\_event",

"webhook": A\_WEBHOOK\_OBJECT,

"created\_by": A\_USER,

"created\_at": AN\_RFC3339\_TIMESTAMP,

"trigger": AN\_EVENT\_NAME,

"source": A\_BOX\_OBJECT

}

###### To handle a Webhook notification the application must process HTTP POST requests sent to the notification URL defined for each active Webhook. The requests that we receive will be structured according to the description in the previous section.

###### It is possible to configure Webhook to use **signature** to protect them against attacks. When we configure a Webhook to use a signature, Box generates a cryptographic digest of the notification’s body and attaches it in header. When our application receives the notification it can compute the same digest and compare it to the one attached to the message.

###### In order to attach signatures to an application’s notifications we must first generate **signature key** for the application. Each application has two signature keys in order to support signature rotation.

###### To configure our application’s key we must to be logged in the developer console and go to the Webhook v2 section of the page of the application. We must to go to the “**Generate primary key**” and “**Generate secondary key**”. We must to click each button to generate the key that will be used to compute digests of our application’s notification message. (For more details see [https://docs.box.com/reference #signatures](https://docs.box.com/reference%20#signatures)).

For more information see: <https://docs.box.com/v2.0/docs/getting-started-with-webhooks-v2> or <https://docs.box.com/v2.0/docs/webhooks>

## Annexe

### Box Errors

A representative sample that shows the vast majority of the statuses and messages that we can expect to see if we build integration with Box.

|  |  |  |
| --- | --- | --- |
| **Status** | **Error Code** | **Message** |
| **400** | *bad\_request* |  |
| *item\_name\_invalid* | Item name invalid |
| *terms\_of\_service\_required* | User must accept custom terms of service before action can be make |
| *requested\_preview\_unvailable* | requested preview unavailable |
| *folder\_not\_empty* | Cannot delete – folder not empty |
| *invalid\_request\_parameters* | Invalid input parameters in request |
| *user\_already\_collaborator* | User is already a collaborator |
| *cannot\_make\_collaborated\_subfolder\_private* | Connote move a collaborated subfolder to a private folder unless new owner is explicitly specified |
| *item\_name\_too\_long* | Item name too long |
| *collaborations\_not\_available\_on\_root\_folder* | Root folder cannot be collaborated |
| *sync\_item\_move\_failure* | Cannot move a synced item |
| *requested\_page\_out\_of\_range* | Requested representation page out of range |
| *cyclical\_folder\_structure* | Folder move creates cyclical folder structure |
| *bad\_digest* | The specified Content-MD5 did not mach what we received |
| *invalid\_collaboration\_item* | Item type must be specified and set to “folder” |
| *task\_assignee\_not\_allowed* | Assigner does not have sufficient privileges to assign task to assignee |
| *invalid\_status* | You can change the status only if the collaboration is pending |
| **401** | *unauhthorized* | Unauthorized |

|  |  |  |
| --- | --- | --- |
| **403** | *forbidden* |  |
| *storage\_limit\_exceeded* | Account storage limit reached |
| *access\_denied\_insufficient\_permissions* | Access denied – insufficient permission |
| *access\_denied\_item\_locked* | Access denied – item locked |
| *file\_size\_limit\_exceeded* | File size exceeds the folder owner’s file size limit |
| *incorrect\_shared\_item\_password* |  |
| *access\_from\_location\_blocked* | You’re attempting to log in to Box from a location that has not been approved by your admin. Please talk to your admin to resolve this issue |
| **404** | *not\_found* |  |
| *preview\_cannot\_be\_generated* | Preview cannot be generated |
| *trashed* | Item is trashed |
| *not\_trashed* | Item is not trashed |
| **405** | *method\_not\_allowed* | Method not allowed |
| **409** | *item\_name\_in\_use* | Item with the same name already exists |
| *conflict* | A resource with this value already exists |
| *user\_login\_already\_used* | User with the specified login already exists |
| *recent\_similar\_comment* | A similar comment has been made recently |
| **412** | *sync\_state\_precondition\_failed* | The resource has been modified. Please retrieve the resource again and retry |
| *precondition\_failed* | The resource has been modified. Please retrieve the resource again and retry |
| **429** | *rate\_limit\_exceeded* | Request rate limit exceeded, please try again later |
| **500** | *internal\_server\_error* | Internal server error |
| **503** | *unavailable* | Unavailable |

### Box integration behaviour

Table box integration behaviour 1

|  |  |  |  |
| --- | --- | --- | --- |
| **enableBoxIntegration** | **boxFolderId** | **boxAllowUpdate** | **Behaviour** |
| 0 | null | 0 | Files are always uploaded and downloaded from standard Wigii server (no link to Box) |
| 1 | null | 0 | File can be chosen from Box and linked to field, but not uploaded to Box.  Or File is not chosen from Box but uploaded and stored into Wigii server (standard behaviour) |
| 0 | not null | 0 | Uploaded File is always uploaded to specified Box folder and then linked to Field. If a file with same name already exists in Folder, then a new version is created. Files can only be uploaded never chosen from Box. |
| 1 | not null | 0 | Or an existing Box file can be chosen using the File picker, or a local file can be uploaded into the specified Box folder. File is never stored on the Wigii server. |
| 0 | null | 1 | End user can never display the Box file picker. But if a File has been linked from Box, then file can be updated by uploading a new version from a local file.  Else File is stored on the Wigii server |
| 1 | null | 1 | If a File has been linked from Box, then file can be updated by uploading a new version from a local file.  Else File is stored on the Wigii server |
| 0 | not null | 1 | A new File is always uploaded to Box specified folder.  When uploading a local file, a new version of the File is created in Box |
| 1 | not null | 1 | End user can either choose an existing File in Box and link it and then make it updatable, or upload a new File in the specified Box folder and keep it editable.  File is never saved on the Wigii server. |