XContest API – Gate Flight

This API is capable to claim flight on XContest server. As a trasport mechanism, classical HTTP POST request/response is used. Request is multipart (the same as created by the browser when submitting form with attribute *enctype="multipart/form-data"*). Response body contains JSON with all data needed for next step of processing.

1 API key + shared secret

To use XContest API from within an application (desktop or mobile, etc.) you need unique **API key** and shared secred (**hash**) for your application. Both you can request from XContest team – contact us on info@xcontest.org and also tell us:

- **name** of the app, platform
- **username** of XContest user responsible for the app (not required)

We will send you both API key and shared secret for your app. Shared secret will be a hidden part of your application and should not be shown, sent or be accesible to the users - unlike API key, which is usually part of each http request on XContest API.

2 Gate ticket

Before each *Gate (POST) request*, you must get **Gate ticket** from the server. It means one additional HTTP GET request to the URL:

http://www.xcontest.org/api/gate/ticket/?key={api_key}**&hash=**{hash}

where:

- **{api_key}** => your API key
- **{hash}** => SHA1 hash of the string (**api_key**+**shared_secret**) – of course without brackets and plus sign, just SHA1 hash of concatenation of both

As a response you get simple JSON

- in case of success: **{ticket : "your_gate_ticket"}**
- in case of error: **{error : {message: "Unable to create ticket"}}** (or other message, depends on the error type – these basic errors messages are always in english)

Keep in mind that *Gate ticket* is valid only for the following single *Gate (POST) request*. Ticket validity is 1 hour (in most cases the POST request is immediate). In case another *Gate (POST) request* is needed, you must obtain new *Gate ticket*.

3 Gate (POST) request

In optimal case, only one *Gate request* should be needed to successfully submit the flight. But in other cases two or more requests (steps) will be needed – at least in case of any error or missing required value. Your application must be able to process the response of *Gate request* correctly, ask user to complete or change the data according to the error messages and make another *Gate request*, until the flight is submitted completely (all *Gate requests* needed to submit one flight creates one *session*).

But don't worry, API will guide you:). Response for each *Gate request* contains JSON with data you need – the most important, complete **form definition** with all controls, control types, possible options, current values, validation results, errors and error messages.

You can use the form definition according to your needs – you can parse whole definition of the form and create form UI completely according to it, or maybe you only want to check what's wrong.

3.1 Typical scenarios of use

3.1.1 First scenario: Authenticate pilot first, submit flight then

This scenario requires minimum of 2 *Gate requests* to submit the flight.

First *Gate request* requires only authentication controls (see chapter ***User authentication controls***). In case of successful authentication, response will contain complete definition of "empty" form with all controls. Your application processes that definition and creates and displays form UI to the pilot. Pilot completes and submits the form. Submitting the form generates second *Gate request* with required flight data. In case some error occurs, the form is returned and errors must be solved by the pilot. In case there will be no errors, sessions ends and terminates with success.

3.1.2 Second scenario: Submit all at once

This scenario has advantage of doing only one *Gate request*, if everything goes well. Your application must create initial Gate request according to the description in the chapter Gate request controls. Also in this scenario your application must be able to handle errors and provide the chance to the pilot to solve the errors and complete the form.

3.2 Gate request URL

http://www.xcontest.org/api/gate/request/?flight

&key={api_key} – your API key; required

&ticket={ticket} – your Gate ticket; required

&hash={hash} – SHA1 hash of the string (**ticket+api_key+shared_secret**); required

&authticket={authticket} – plays the role of session identifier – leave empty value (only **&authticket=**) in case of this is the first request within the flight submit session; for following requests with the flight submit session, use the value of **authTicket** property you get from the response of the first request; required

&lng={lng} – ISO language code – language in which you receive texts in the response (labels, error messages etc.); optional, default value is **en**

&autocomplete={Y|N} – if you send the request with autcomplete=N, the flight will not be completely submited in this step (in other words, even if there will be no other errors, which normally leads to successfully completed submit, one error will be triggered); optional, default value is **Y**

3.3 Gate request controls

In the latter text all needed form controls to send.

3.3.1 User authentication controls

Following controls ensure *authentification of the pilot*. They must be part of the initial *Gate request*.

- **login[username]** – username of XContest pilot
- **login[password]** or **login[hash]** – only one control from these couple should be used
    - *login[password]* is plain text password of the Xcontest pilot

- *login[hash]* is SHA1 hash of: md5(**password** in lowercase) +**ticket**+**api_key**+**shared_secret** (keep in mind that you must create *md5 hash of lowercased password* before concatenation with next parts – and then create sha1 hash)

We recommend to use preferably *login[hash]* instead of *login[password]*.

In case the authentification fails, response contains JSON: **{error : {message: "Login user: login for username '{username}' failed"}}**.

IMPORTANT: You **must not** send these controls with subsequent *Gate requests*, if user authentication was successful (if you don't get login error in the response). Authentication must be made only once at the beginning of session in the the initial *Gate request* and is valid for complete session (see also 3.2 – authticket).

3.3.2 Flight controls

- **flight[tracklog]** – IGC tracklog file. Must have **.igc** extension. Alternatively, the IGC file may be added as zip archive. First file from inside the **.zip** archive is considered as the IGC tracklog file (must have .igc extension). This control is required, until successfully uploaded.

  IMPORTANT: After successful upload (in possible next steps/Gate requests within one submit session) the tracklog **must not** be uploaded again (in other words, it is not possible to change the IGC for the flight).

  HTML representation: <input type="file" name="flight[tracklog]">

- **flight[is_active]** – determine if the flight will be active (public, it means visible in all lists and ranking) or not. Possible values are '**Y**' or '**N**'.

  Autocomplete: In case there will be no flight[is_active] value in *Gate request*, default is N (not active). Most likely expected by pilots is, of course, that their flights will be active.

  HTML representation: <input type="checkbox" name="flight[is_active]" value="Y" checked>

- **flight[contests]** – contests to put flight in. Cannot be used in the first *Gate request*, because it must be proccessed by the server and depends on pilot's registrations to the contests and also the tracklog (date, area, etc.).

  Autocomplete: In case there will be no flight[contests] values in *Gate request*, flight will be simply put into each contest, where it should have been, by default.

  HTML representation: <input type="checkbox" name="flight[contests][]" value="1" checked> <input type="checkbox" name="flight[contests][]" value="2" checked>...

- **flight[segment]** - segment(s) detected inside the tracklog. Cannot be used in the first *Gate request*, because it must be proccessed by the server and options are detected according the uploaded tracklog. More segments usually means more flights or at least there are some gaps inside the tracklog.

  Autocomplete: In case there is only 1 segment in the tracklog (most common case), start and end point of the flight are unambiguous. In case there are more segments detected, pilots must always choose one option before flight can be sucessfully submitted.

  HTML representation: <input type="radio" name="flight[segment]" value="1" checked"> <input type="radio" name="flight[segment]" value="2">...

- **flight[class]** – FAI class.

  Options: These values are currently supported ('value' => 'label' pairs):

  - '3' => 'FAI-3 (PG)'
  - '1' => 'FAI-1 (HG)'

    - '5' => 'FAI-5 (Rigid wing)'
    - '2' => 'FAI-2 (Rigid wing)'
    - '11' => 'RPF (Foot-launched)'
    - '12' => 'RPL (Landplane/Trike)'

    Autocomplete: In case there will be no flight[class] value in *Gate request*, server tries to guess FAI class based on the FAI class used by last submitted flights, but result of this method is not certain and reliable. If you like to avoid error messages, your application should send this value explicitly.

    HTML representation: <select name="flight[class]">...

- **flight[glider_name]** – full name of the glider.

  Autocomplete: In case there will be no flight[glider_name] value in *Gate request*, server first tries to extract glider name from the IGC header. If still no glider found, server tries to set last used glider (the condition is that such glider was used by at least 3 consecutive previous flights of the pilot). Paragliders (FAI class = 3) are searched in our database, if not found, glider is considered as custom glider.

  Options: Server creates options list of all gliders used by pilot with conjunction of the defined FAI class. Gliders are sorted according to the average age of the flights with each used glider, from "newest" to "oldest". Keep in mind that the value of flight[glider_name] may differ from these generated options (= pilot can use new/custom glider for the flight submitted).

  HTML 5 representation: <input type="text" name="flight[glider_name]" list="glider-list"> <datalist id="glider-list"><option value="AXIS Venus 3">...</datalist> (alternatively, in classic HTML 4 it could be combo of text input and select list, with possible javascript treatment)

- **flight[glider_catg]** – category of the paraglider. This control is required (and used) only if FAI class = 3 and the glider is considered as custom one.

  Options: These values are currently supported ('value' => 'label' pairs):

    - 'T' => 'tandem/biplace'
    - 'A' => 'EN-A/LTF 1'
    - 'B' => 'EN-B/LTF 1-2'
    - 'C' => 'EN-C/LTF 2'
    - 'D' => 'EN-D/LTF 2-3'
    - 'Z' => 'open/proto'

  You may notice that EN/LTF categories are mixed with one tandem category (so doesn't matter what EN/LTF rating the tandem glider has). Please respect the order of these options in your application, otherwise tandem pilots would tend to choose EN/LFT rating of their gliders instead.

- **flight[country]** – country of the takeoff.

  Autocomplete: Server always tries to detect the country automatically. Detection is closely related to the number of segments in the tracklog. In case there are more segments, detection is done for each segment. If there is only one country detected, detection is considered unambiguous and such country is taken. If there are more options, pilot is forced to choose one.

  Options: All possible detected countries are included into server-generated options list. In case the detection fails, all countries are offered in options list. But in most cases, there is only one option, unambiguously detected country.

- *HTML representation:* <select name="flight[country]"><option id="57">Czech republic [CZ]</option>...</select>
- **flight[takeoff]** or flight[takeoff_name] - takeoff from our database, or custom one (if the takeoff is not in our database).

  Autocomplete: flight[takeoff] is auto-detected and in case it is close enough to one the takeoffs in our database, it is considered as unambiguously detected. If the takeoff is not detected or there are no takeoffs in our database within 20 km range, server tries to complete the flight[takeoff_name] from the IGC file header.

  Options: All takeoffs from our database closer than 20 km from the detected takeoff point of the tracklog are included into server-generated options list.

  HTML representation: <select name="flight[takeoff]"><option value="164">Krupka ~3.52km</option>...</select> <br> <input type="text" name="flight[takeoff_name]">
- **flight[comment]** and **flight[url]** – optional controls. Comment of the pilot and URL he/she wish to add to the flight detail (with more photos, video, blog entry, etc.)

3.3.3 Overview of flight form controls

Form control name	Type	Required	In first Gate request	Autocompleted
flight[tracklog]	TYPE_FILE	Y	Y	
flight[is_active]	TYPE_LOGICAL	Y	Y	Y
flight[contests]	TYPE_CHOOSE_MORE	Y		Y
flight[segment]	TYPE_CHOOSE_ONE	Y		Y* (if unambiguos)
flight[class]	TYPE_CHOOSE_ONE	Y	Y	Y* (from last flights)
flight[glider_name]	TYPE_TEXT_CHOOSE_ONE	Y	Y	Y* (from IGC)
flight[glider_catg]	TYPE_CHOOSE_ONE	Y*	Y* (if custom paraglider)	
flight[country]	TYPE_CHOOSE_ONE	Y	Y* (not neccesary)	Y* (if unambiguos)
flight[takeoff]	TYPE_CHOOSE_ONE	Y*		Y* (if unambiguos)
flight[takeoff_name]	TYPE_TEXT	Y*	Y* (if custom takeoff)	Y* (from IGC)
flight[comment]	TYPE_TEXT		Y	
flight[url]	TYPE_TEXT		Y	

Legend:

Type *– see Form controls definition*

Required *– which form controls are required for successful flight submit,* ***Y**** *- form control is required conditionaly, depends on the value of another form control;*

In first Gate request *– which form controls may be part of the first (initial) Gate request,* ***Y**** *- may be in the first initial Gate request, but depends on another condition (see text in brackets);*

Autocompleted *– which form controls are autocompleted on the server,* ***Y**** *- autocompleted* *conditionaly (see text in brackets)*

3.4 Gate request response

The body of the response always contains JSON data.

In case of some basic error occurs, like authentication failure or other common API errors, there is only a simple error data structure:

```
{error : {message: "Login user: login for username '{username}' failed"}}
```

In case there are only errors and warnings related to the form processing or there are no errors at all, response body contains this basic datastructure:

```
{
	success: true,
	form: { ... },
	authTicket: "28c59d90c7323dc6bab6",
	authData: { ... }
}
```

where:

- **success** determines if the processing of the form was successful.

  If the value is **false**, some errors occurred during the processing of the form and these errors must be solved by the pilot. If the value is **true**, is means the form is either in **initial phase** (no values for flight controls were sent – for example if only authentication control values were sent) or in **final phase** (flight was successfully submitted). To determine which phase is it exactly, see **form.phase.**

- **form** is the largest object data structure with all definitions of controls (explained later).

- **authTicket** is the session identifier assigned to your flight submit session. This identifier is included only in the response of the first *Gate request* within the flight submit session and your application will use it for all consequent *Gate requests* until the end of the session (successful submit of the flight).

- **authData** is the object with basic data of authenticated user, mainly **id**, **username**, **firstName** and **lastName**.

3.4.1 Form definition

Only important attributes are stated and commented.

```
form: {
	isValid: true,
	phase: 1,
	controls: { ... },
	extinfo: { ... }
}
```

where:

- **isValid** is global info about form and controls validity. In fact it plays the same role as global **success** attribute (in this API version you get the same info from both), but in the future their role may differ (**isValid** is related to the form itself, **success** is related to the *Gate request* as a whole).

- **phase** may have following values:
  - **1** – initial phase. Represents “empty” form, no value of flight controls was sent
  - **2** – process phase. Represents the state “data were received, but errors occured”
  - **3** – final phase. Successful (finished) flight submit, session is terminated.
- **controls** – object with all needed form controls (explained later). Order of controls is defined to be optimal for UI (in other words, pilot should see UI components related to each form control in such order).
- **extinfo** – object with various usefull informations. In final phase (form.phase = 3) contains **flightId** and **flightUrl**, related to the recently submitted flight (your application should respect the fact that the flight may not be accesible immediatelly on given URL – usually it requires no more than half a minute, but it depends on actual tracklog and server load).

3.4.2 Form Control definition

First example of “comment” form control definition.

```
{
        ...,
        comment : {
                ident: "comment",
                type: "TYPE_TEXT",
                isRequired: false,
                isValid: true,
                id: "flight-comment",
                name: "flight[comment]",
                value: "",
                label: "Komentář (popis letu)"
        },
        ...
}
```

where:

- **ident** si internal identifier (the same as the key in the controls object)
- **type** is one of following:
  - **TYPE_TEXT** – simple text field, like <input type=”text’”> or <textarea> in HTML
  - **TYPE_LOGICAL** – true or false, like <input type=”checkbox”> in HTML
  - **TYPE_CHOOSE_ONE** – select from predefined options list, like <select> (drop-down list) or set of <input type=”radio”> elements in HTML
  - **TYPE_CHOOSE_MORE** – select from predefined options list with possibility to choose more options, like <select multiple> (list box) or set of <input type=”checkbox”> elements in HTML
  - **TYPE_TEXT_CHOOSE_ONE** – combined select with textbox (combo box) like <input type=”text” list="options-list"> in HTML 5
  - **TYPE_FILE** – field for file upload, like <input type=”file”> in HTML
- **isRequired** is boolean attribute - if the value of the form control is required or not for correct submission

- **isValid** is similar as **form.isValid**, but only in context of this form control. If the control is not valid, there is something wrong with (at least it is required but no value, etc.). Usually you get also relevant error message (explained later).
- **id** – unique identifier, main relevance is for HTML(id attribute)
- **name** – form control name
- **value** – actual form control value. Usually text string, but for TYPE_CHOOSE_MORE it is an array(!). For TYPE_LOGICAL it is always 'Y', use **checked** attribute to find out boolean (true/false) value instead.
- **label** – label, instruction related to the form control (some controls may not have labels

3.4.2.1 Type-dependent attributes

TYPE_CHOOSE_ONE, TYPE_CHOOSE_MORE, TYPE_TEXT_CHOOSE_ONE:

- **options** – array of options, each option is an object (**id**, **value** and **selected** correspond to the HTML attributes of the same names, **text** would be usually the content of option element)

  { id : "...", value : "...", text : "...", selected : true/false }

- **emptyValueLabel** – label used for empty value (option with value "")

TYPE_LOGICAL:

- **checked** – true/false

3.4.3 Errors, warnings and messages

Errors are events which effectively stop form proccessing and causes the forms remains in process phase (form.phase = 2). **Warnings** are events which generate informations, but do not stop the processing of the form (Flight may be successfully submitted even with warnings). **Messages** are understandable text messages for both errors and warnings.

In **form control** definition they are present only if needed:

```
tracklog : {
        ...,
        errors : {
                igcNotUnique : "igcNotUnique"
        },
        warnings : {
                igcValidity : "invalid"
        },
        messages : [
                {
                        type : "error",
                        code : "igcNotUnique",
                        label : "This IGC file have been allready submitted to the XContest
system. Probably you are trying to claim your flight again. If you have claimed your
flight before a short time, processing of your flight is still running. If you like to change
flight comment, flight properties or add/edit photos, you should go to the page My
flights, where you can edit each single flight."
                },
                {
                        type : "warning",
                        code : "invalid",
                              label : “WARNING: IGC file was NOT validated successfully. You can claim your flight and you will see it in Flights-list and Daily-score, however it will not be scored in any ranking categories.”
                    }
          ]
}
```

This couple of error/warning cannot appear together in reality, but it is a good example how it works in principle. Keep in mind that **errors** and **warnings** are objects and **messages** is an array. Also **messages[n].code** attribute is related always to the value of the error or warning (“igcUnique” and “invalid”, in our example).

Errors, warnings and messages may also on the level of **form** definition, the system is exactly the same. One important error on the form level is **autocompleteSupressed** – this error may is triggered each time when GET attribute **autocomplete=N** is used in URL of Gate request and no other error occurs.

Just notice that not all errors and warnings must have associated message.

3.4.4 Alternatives

Used for functional dependency of two or more form controls. Form control may have one or more **alternatives** (and such form controls are **alternativeFor** the master form control). **Alternative** is form control which is used only in case the master form control's value is not within the values of options list.

There are two alternative relations in Flight form definition:

- **flight**[**glider_name**] → **flight**[**glider_catg**] – glider_catg is used only if the glider is not selected from the options list (custom glider)
- **flight**[**takeoff**] → **flight**[**takeoff_name**] - takeoff_name is only if the takeoff is not selected from the options list (custom takeoff)