Author and Copyright

- Authored by Brett Bender brett@liquidplanner.com

- Copyright © 2012 LiquidPlanner, Inc.

- Revised: $Date: 2012-03-28 14:24:23 -0700 (Wed, 28 Mar 2012) $

Introduction

Overview

This document describes how to integrate with the LiquidPlanner API, which

- is found at https://app.liquidplanner.com/api
- uses HTTP Basic Auth over SSL to authenticate each request
- generally follows RESTful resources conventions
- represents resources as JSON

Prerequisites

The intended audience for this document is software developers interested in integrating with LiquidPlanner. We assume that you already have a LiquidPlanner account and some familiarity with the app. If not, please check out the LiquidPlanner workspace guide.

Create a Sandbox Workspace

We recommend that you create a new LiquidPlanner workspace for your initial API experimentation and testing. This way you won't disturb your production data (or other users of your production workspace), you can work with a small and controlled set of data, and you can easily clean up (or start over) if things go sideways.

Add a few projects and tasks (so that there's something to see, but not too much to follow) to your sandbox workspace.

Your First API Request

Let's start with a simple "Hello, World!" request that fetches your LiquidPlanner account details.

Throughout this document, we'll provide examples using curl. We use curl because it is available across platforms and assumes no knowledge of any particular programming language.

Examples will look like:

```
% command arg1 arg2 ... argN
... command output ...
```

where the "%" is your command prompt; long command lines may be continued to the next line by ending the line with a backslash. Type the command followed by its arguments, and hit *enter*. You should get output similar to that in the example (though we've reformatted, truncated, or omitted example output for readability).

Store Defaults in a Config File

Create a config file similar to the following, using your login information:

```
% cat ~/.liquidplanner_curl
compressed
user: api_user@example.com:api_password
-H "Content-Type: application/json"
write-out: "\nStatus: %{http_code}\n"
```

This tells curl to ask for and to handle a compressed response, provides your credentials, specifies that POSTed data is JSON-encoded (rather than form-encoded), and instructs curl to output the HTTP status code after the response body (which is useful for troubleshooting).

To keep your password secure, make the config file readable only by you (using chmod on Linux, Mac OS X, etc.):

```
% chmod 600 ~/.liquidplanner_curl
```

On Windows, use the file properties pane (or the "cacls" command).

Alternatively, omit the password from the file (including only your email address), and you'll be prompted for it each time you run curl.

Run the Request

Now tell curl to read configuration from this file, and request your account info:

```
% curl -K ~/.liquidplanner_curl https://app.liquidplanner.com/api/account
{
  "avatar_url": "/images/avatars/none.png",
  "created_at": "1977-09-10T00:00:01+00:00"
  "created_by": 23,
  "email":      "api_user@example.com",
  "first_name": "Brett",
  "id":         23,
  "last_name":  "Bender",
  "timezone":   "US/Pacific",
  "type":       "Member",
  "updated_at": "2010-01-01T00:00:01+00:00",
  "updated_by": 23,
  "user_name":  "brett",
}
Status: 200
```

For brevity in subsequent examples, we write *curl* where we really mean:

```
curl -K ~/.liquidplanner_curl
```

so that your configuration file is used. You may want to create a command alias such as *lpcurl* for the above; it will save you some typing.

Examples

In the following examples, we use :id in the request URLs as a placeholder, and you should substitute values specific to your data (your workspace ID, the ID of one of your tasks, etc).

List the Contents of the Workspace

The contents of a workspace are organized in a tree. Every workspace has a root. The root may be empty (in which case you have a empty workspace), or it may contain projects and packages. In turn, these projects and packages may contain other items.

To fetch only the root, request treeitems with a depth of 0:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/treeitems?depth=0
... wall of text ...
```

If you want the entire tree, you do:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/treeitems?depth=-1&leaves=true
... larger wall of text ...
```

See The Workspace Tree for details.

Add a Task

To create a task, you:

- POST to the /tasks resource of the workspace
- With a parameter named “task” whose value is a hash of task attributes
- With task attributes including at least “name” (throw in a folder_id and package_id if you’re feeling adventurous)

Supposing we want to add a task “learn the API”:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks \
       -d '{"task": {"name": "learn the API"}}'
{
  "name": "learn the API",
  "id":   79104
  ...
}
Status: 201
```

If you get an error here, double-check that you've set the Content-Type of your request to "application/json" using your curl configuration file.

Comment on a Task

To create a comment, you:

- POST to the …/comments resource of an item
- With a parameter named "comment" whose value is a hash of comment attributes
- With an attribute "comment" in the hash whose value is the text of the comment

Supposing we want to comment "working on it…" on our task "learn the API":

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id/comments \
       -d '{"comment": {"comment": "working on it..."}}'
{
  "comment": "working on it...",
  "id":       10941,
  "type":     "Comment",
  "item_id":  79104,
  ...
}
Status: 201
```

Mark a Task Done

To update a task, use the PUT HTTP method and specify the task ID.

You have now learned the fundamentals of the LiquidPlanner API, so mark the task done:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id \
       -X PUT -d '{"task": {"is_done": "true"}}'
{
  "name":    "learn the API",
  "id":      79104,
  "is_done": true,
  ...
}
Status: 200
```

Convenience Methods

Certain operations that are difficult or inefficient to perform using a strictly RESTful interface have convenience methods provided.

Uploading and Downloading Documents

To create a new document, do a multipart/form-data POST with parameters:

document[file_name]
: name of the file, to be used in download links

document[description]

description of the file

document[attached_file]
: file data (as from a file upload control in an HTML form)

An example, assuming you have a file ‘requirements.xls’ to upload:

```
% curl http://app.liquidplanner.com/api/workspaces/:id/tasks/:id/documents \
       -F 'document[attached_file]=@requirements.xls'                      \
       -F 'document[file_name]=requirements.xls'
```

This action is an exception to the general rule that submitted data should be JSON-encoded; multipart/form-data supports a binary encoding, and is therefore more efficient on both network and CPU than submitting the attached_file as e.g. a Base64 encoded string in JSON.

To download an existing document, do a GET with ‘download’ or ‘thumbnail’ appended, like:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id/documents/:id/download
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id/documents/:id/thumbnail
```

Updating Estimates and Tracking Time

You can update your estimate, track hours worked, and add a comment with a single POST using the *track_time* method on tasks.

The following request tracks four hours of work on the task with ID 79104 in the workspace with ID 1, and sets the remaining estimate to a low of 1.5 days and a high of 3 days.

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id/track_time \
       -d '{ work: 4, low: "1.5d", high: "3d" }'
```

Parameters include:

member_id
: id of the member to track work against and comment as; optional, defaults to you (the current user)

work
: hours worked; optional

activity_id
: id of the activity for the work; required if you are using timesheets and provide a value for work

low
: low estimate for remaining work, hours (or a string such as “4h”, “0.5d”); usually optional, must be provided if you provide high, and omitted or set to ‘0’ when setting is_done

high
: high estimate for remaining work, hours (or a string such as “4h”, “0.5d”); usually optional, must be provided if you provide low, and omitted or set to ‘0’ when setting is_done

is_done
: if true, marks task as done; low and high must each be omitted or explicitly zero

done_on
: datetime on which the task was done; valid only when is_done=true, defaults to now

work_performed_on
: date on which the work was performed, default today

comment

comment to add to the task, optional

Workspace Comment Stream

You can fetch a list of recent comments (matching those displayed on the web dashboard) using the *comment_stream* method on workspaces. The item commented upon is embedded in the comment.

To get (up to) 100 comments within the previous week, do:

```
% curl https://app.liquidplanner.com/api/workspaces/:id/comment_stream
```

Optional parameters include:

for_me
: if *true* then only return comments directed at me or on owned or watched items, default false % curl https://app.liquidplanner.com/api/workspaces/:id/comment_stream?for_me=true exclude comments made by me, default true % curl https://app.liquidplanner.com/api/workspaces/:id/comment_stream?ignore_mine=false start of recent period, default 7 days before the end_date % curl https://app.liquidplanner.com/api/workspaces/:id/comment_stream?start_date=2011-06-02 end of recent period, default now % curl https://app.liquidplanner.com/api/workspaces/:id/comment_stream?end_date=2011-06-05

limit
: max number of comments to return, default 100 % curl 'https://app.liquidplanner.com/api/workspaces/:id/comment_stream?end_date=2011-06-05&limit=1'

Starting and Stopping Timers

You can request a list of your timers in a given workspace:

```
GET /api/workspaces/:workspace_id/timers
```

and to access your timer for a particular task:

```
GET /api/workspaces/:workspace_id/tasks/:task_id/timer
```

This will always return a timer object (either a running/stopped timer, or a newly created "stub" timer – but never nil).

To start, stop, or clear a timer:

```
POST /api/workspaces/:workspace_id/tasks/:task_id/timer/start
POST /api/workspaces/:workspace_id/tasks/:task_id/timer/stop
POST /api/workspaces/:workspace_id/tasks/:task_id/timer/clear
```

On success, each of these calls returns the affected timer.

RESTful Resources

For background on RESTful APIs in general, see Wikipedia or another suitable reference.

HTTP Methods

If you want to …	Then you should …
fetch all instances of the resource	GET /api/resource
fetch an instance of the resource	GET /api/resource/:id
create a new resource	POST /api/resource
update an existing resource	PUT /api/resource/:id
delete an existing resource	DELETE /api/resource/:id

If your HTTP client does not implement the PUT and DELETE methods, then you can use a POST with an additional parameter named *_method* and a value of either *PUT* or *DELETE*.

For example, to delete a task using the POST method rather than the DELETE method, do:

```
POST /api/workspaces/:id/tasks/:id?_method=DELETE
```

Scoping / Nesting of Resources

Resources (except for your account and workspaces themselves) are scoped by or nested within a workspace. This scoping is represented in the request path, so to list the tasks in a given workspace, do:

```
GET /api/workspaces/:workspace_id/tasks
```

and to show a single task, do:

```
GET /api/workspaces/:workspace_id/tasks/:task_id
```

This nesting pattern may be repeated, as with the comments for a task:

```
GET /api/workspaces/:workspace_id/tasks/:task_id/comments
```

Resource Parameter

When creating or updating a resource, the API requires a request parameter with the same name as the resource (in the singular form), whose value is a hash of the resource's attributes.

Putting together the HTTP methods, resource nesting, and the resource parameter, we have the following:

If you want to…	Then you should…	To…	With a parameter…
create a task	POST	/api/workspaces/:workspace_id/tasks	task = …*hash*…
create a comment on a task	POST	/api/workspaces/:workspace_id/tasks/:task_id/comments	comment = …*hash*…
update a folder	PUT	/api/workspaces/:workspace_id/folders/:folder_id	folder = …*hash*…

LiquidPlanner Resources

You can reference the available resources at:

https://app.liquidplanner.com/api/help/urls

The following list represents the relationship (including nesting) of the various resources available via the API:

- Workspaces
    - ♦ Members
    - ♦ Activities
    - ♦ Tasks
        - ◊ Activities
        - ◊ Note
        - ◊ Timer
        - ◊ Comments
        - ◊ Documents
        - ◊ Links
        - ◊ Dependencies
        - ◊ Estimates
        - ◊ Snapshots
        - ◊ TimesheetEntries
    - ♦ Folders
    - ♦ Packages
    - ♦ Clients
    - ♦ Projects
    - ♦ Events
    - ♦ Milestones

Most resources nested on Tasks (including Note, Activities, Comments, Documents, Links, and Dependencies) are also found on Folders, Packages, Projects, Events, and Milestones.

TimesheetEntries are found only on Tasks, Events, and Milestones.

Timers are found only on Tasks, Events, and Milestones.

Estimates on Tasks are created directly from user input, while those on Events are calculated from the event duration, and those on Milestones are fixed at zero for low and high effort.

Estimates on Folders, Packages, and Projects are derived from their contained items.

Technical Reference

Base URL

All API requests are made relative to this base URL:

```
https://app.liquidplanner.com/api
```

The base URL may change (for example, it could move to a separate hostname dedicated for use with the API); use a configuration setting or symbolic constant for it in your code.

Authentication and Authorization

Each API request is authenticated by HTTP Basic Authentication. Use the same credentials (email address and password) that you use to login to the LiquidPlanner web application.

Incorrect or missing authentication credentials will return status 401.

Once authenticated, you are authorized to access information only from the workspaces in which you are an active member. If your membership in a workspace is inactive or the workspace owner disables API access to the workspace, then the workspace will not appear in your list of workspaces, and you will not be able to access its contents.

Types, Attributes, Data Model

The available record types and their attributes may change over time; you can view a dynamically generated reference at https://app.liquidplanner.com/api/help/types

The Workspace Tree

Items and Containers

We use the term *item* or *treeitem* to refer to the things that you can arrange in your workspace tree – treeitems include Tasks, Events, Milestones, Packages, Projects, and Folders.

We use the term *container* to refer to items that may contain other items – containers include Roots, Packages, Folders, and Projects.

We use the term *leaf* to refer to items that cannot contain other items – leaves includes Tasks, Events, and Milestones.

Tree Representation in JSON

The tree hierarchy is represented in JSON using a *children* attribute, whose value is an array of child hashes. Each child hash may (if it is a container) also have a ‘children’ attribute.

```
{ "children": [ {},
                { "children": [ {},
                                {},
                                {}
                              ]
```

```
                    },
                    {}
                ]
}
```

This is the default behavior for listing the tree via the treeitems resources. If you want only a flat array (with the parent / child relationship indicated solely by corresponding IDs and not by the nested hash structure of the data), specify “flat=true” as a query parameter.

Note that if you both:

a. specify a filter and b. do not specify that you want context

then we automatically force flat=true (because we cannot return a tree if your filter matches a child but not its parent, and you don’t want context).

Including Ancestors and Children

When fetching a treeitem by ID from the treeitems resource, you may specify these optional parameters:

depth
: 0 (default, only the item itself), -1 (for all depths), or a positive number N (for depths no greater than N)

leaves
: *false* (the default, including only structure / containers) or *true* (including everything)

  % curl
  https://app.liquidplanner.com/api/workspaces/:workspace_id/folders/:client_id?include_children=all
  … the client, and the projects and folders for that client …

  % curl
  “https://app.liquidplanner.com/api/workspaces/:workspace_id/folders/:client_id?include_children=all&include
  … everything above, plus tasks, events, and milestones …

item_context
: *false* (the default, only the specified item), or *true* (include the ancestors of the item, up to the root)

filter_context
: *false* (the default, only the specified filter matches), or *true* (include the ancestors of the filter matches, up to the root)

Use *false* for the context parameters where e.g. you are “zoomed to” or have “opened” the item and only need information about the item itself, and *true* for e.g. rendering the placement of the item in the tree.

A Few Simple Rules for Arranging Items in the Tree

Actually, it’s arguable whether they are simple or few… but here there are.

- Every item has a parent (except the root).
- A package’s parent must be a package or the root.
- A project’s parent must be a package or the root
- A folder’s parent must be a project or a folder.
- A leaf’s parent must be a package, a project, or a folder.

- Only leaf items may be packaged (have their package_id and package_priority set)
- A leaf may be packaged only if it its parent is not a package (i.e. its parent must be a project or a folder)

Ordering of Items in Containers

Every item has a ‘global_priority’ attribute.

Packaged items additionally have a ‘global_package_priority’ attribute. For scheduling purposes, this overrides the item’s global_priority.

The values of these attributes are arrays of numbers specifying relative positions in the tree. The values of these attributes are comparable or sortable, but otherwise arbitrary; e.g. you cannot infer the quantity of a project’s items from the global_priority of its lowest priority item.

Two items with the same parent will have identical values for global_priority in all but the last position in the array, and the value in this last position indicates their relative order within the container itself (lesser value first, greater value second). The same principles apply to two items packaged in the same package and the values of their global_package_priority attributes.

Moving Items

Every item (except the root) has a parent container.

To move an item into a container without specifying its position in the container, simply update the item’s parent_id. This will move the item into the specified container in last position.

To specify an item’s position within a container, POST to one of the following:

- move_before
- move_after

with a parameter:

other_id
: the ID of the other item, before or after which to move this item

for example:

```
POST /api/workspaces/:workspace_id/tasks/:id/move_before?other_id=:id
POST /api/workspaces/:workspace_id/tasks/:id/move_after?other_id=:id
```

or, where applicable:

packaged_other_id
: the ID of the other item (in its packaged location!), before or after which to move this item

for example:

```
POST /api/workspaces/:workspace_id/tasks/:id/move_before?packaged_other_id=:id
POST /api/workspaces/:workspace_id/tasks/:id/move_after?packaged_other_id=:id
```

Packaging Items

Leaf items may be both moved and packaged, while container items may only be moved (but not packaged).

When you package an item, its package_id is set to that of the containing package, but its parent_id remains the same. Its effective priority for scheduling becomes that of its packaged location (whether this is higher or lower than the priority of its “real” location).

You can only package an item if its parent is not a package (i.e. when its parent is a folder or a project).

To specify an item’s packaged position within a container, POST to one of the following:

- package_before
- package_after

other_id
: the ID of the other item, before or after which to package this item

  POST /api/workspaces/:workspace_id/tasks/:id/package_before?other_id=:id POST /api/workspaces/:workspace_id/tasks/:id/package_after?other_id=:id

Note that you never use the *packaged*other_id_ here, because it is unambiguous which other you mean (either its package_id is set, or its parent_id is that of a package – but never both).

Treeitems Resource

There is a special resource for treeitems; you can use it to “blindly” show, update, or nest into any sort of item without knowing its type in advance (but not to create new items – use the standard type-specific resources).

```
GET /api/workspaces/:workspace_id/treeitems/:item_id
PUT /api/workspaces/:workspace_id/treeitems/:item_id
GET /api/workspaces/:workspace_id/treeitems/:item_id/comments
etc.
```

Custom Fields

You can configure custom fields for your projects and tasks. The custom fields for a given item are returned as a hash, and are settable in the same way. Suppose you have a custom field “Urgency” with values of “High” and “Low”. Then if you get a task whose Urgency is High, you will see:

“custom_field_values”:{“Urgency”:”High”}

and you can change the Urgency to Low with a PUT that includes:

“custom_field_values”:{“Urgency”:”Low”}

as one of the attributes.

Including Associated Records

Specify an 'include' parameter with a comma separated list of things to include. Valid things to include are:

- activities
- comments
- dependencies
- dependents
- documents
- estimates
- links
- note
- snapshots
- timer

For example, getting a specific task with its comment, documents, links, and note (roughly the information needed for the "Collaborate" page):

```
% curl https://app.liquidplanner.com/api/workspaces/:id/tasks/:id?include=comments,documents,link
```

Filtering Items

When requesting items such as folders, packages, or tasks you can filter or restrict the set of results returned to those matching the criteria you specify.

There are two reasons to filter on the server (rather than iterating over the records client-side):

- you can reduce the size of the response by including only matching records
- logic is identical to that used by the web application

To filter, provide a parameter named 'filter' whose value is an array of one or more valid filter strings. The query string should look like:

```
?filter[]=filter_string_1&filter[]=filter_string_2&...
```

By default, the filter conjunction is assumed to be 'AND', such that only those items matching all supplied filters will be returned. If you specify filter_conjunction=OR, then items matching any of the supplied filters will be returned.

Each filter string has three parts:

- attribute
- operator
- value

Examples of filter strings:

```
owner_id = 23
is_done is false
is_on_hold is false
name starts_with BUG REPORT
```

You only need a space between the parts if they would otherwise “run together”, and any extra spaces are ignored. If the value has significant leading or trailing whitespace, throw single or double-quotes around it.

The best practice is to always put a space between the parts, and always put quotes around the value part. But if you forget, we try to be forgiving.

For example, these are all valid and mean the same thing:

```
owner_id=23
owner_id =23
owner_id= 23
owner_id = 23
```

But the following are invalid:

```
namestarts_withBugReport
name starts_withBugReport
namestarts_with BugReport
```

and require whitespace:

```
name starts_with BugReport
```

Single or double-quote the value if it contains significant leading or trailing whitespace. For example:

```
name contains "Bug Report  "
```

will only match items whose name contains “Bug Report ” (where it is followed by two spaces). Without the quotes, the trailing spaces would be ignored (and you’d match any “Bug Report”, with or without spaces after it).

Available Filters

ID Filters

Use the operators ”=” and “!=” to match those items that do or do not have a specific ID set. The value should be an ID number; in the case of ‘owner_id’, ‘created_by’, and ‘updated_by’, you can also use the magical value ‘me’. The values ‘everyone’ and ‘unassigned’ are usable for ‘owner_id’.

- client_id
- created_by
- owner_id
- package_id
- project_id
- updated_by

Boolean Filters

The operator for boolean filters is always “is”, and the value is either “true” or “false”.

- has_alert
- has_an_activity

- has_comments
- has_dependencies
- has_documents
- has_reference
- is_done
- is_in_a_project
- is_on_hold
- is_packaged
- is_shared
- needs_update

String Filters

Operators are ”=” (exact match, case sensitive), “starts_with” (case-insensitive prefix match), and “contains” (case-insensitive substring match).

- name
- reference

Date Filters

The are several operators available for date attributes; not all are available for all attributes.

These are the operators:

Operator	Meaning
within	attr is within N days of now, in the past or future
not_within	attr is not within …
in_next	attr falls before N days after now
never	attr is not set; value is ignored
before	attr is before a specific date
after	attr is after a specific date

and these are the attributes along with their applicable operators:

Attribute	Operators
last_estimated	all
date_done	all
expected_finish	all
earliest_start	all
last_updated	before, after, within, not_within
created	before, after, within, not_within
promise_by	before, after, in_next, never
delay_until	before, after, in_next, never

Ordering and Limiting Lists of Tasks

Order Param

You can request that a list of tasks be returned in a particular order.

Specify an *order* parameter whose value is one of the following:

earliest_start
: default, returns the items with the earliest earliest_start first

updated_at
: returns the most recently updated_at items first

Invalid values return an error response.

Limit Param

You can request that no more than a certain number of tasks be returned.

Specify a *limit* parameter whose value is an integer greater than zero.

Invalid values return an error response.

Request Throttling

Each user account may make up to **30 requests per 15 seconds**. We may tune this limit as needed to maintain overall system availability and responsiveness.

If you exceed this limit, then subsequent requests in the same period may receive a status 503 response. You should wait for the number of seconds specified by the *Retry-After* response header before retrying the request.

Example of the thirty-first request within 15 seconds (using the -i parameter to curl to display response headers):

```
% curl -i https://app.liquidplanner.com/api/account
HTTP/1.1 503 Service Unavailable
Connection: Keep-Alive
Content-Type: application/json; charset=utf-8
Retry-After: 15
{
  "message": "Request throttled, try again later.",
  "error":   "Throttled",
  "type":    "Error"
}
Status: 503
```

API Versioning

You may use the request header *X-API-Version* to specify your expected API version. The currently supported versions are:

```
3.0.0
```

If the version header is omitted, then by default the most recent version of the API will be used to process your request. If this default version is not backwards-compatible with the (presumably older) version that you expect, then the request processing and response may be unexpected or erroneous.

If the version header is supplied and matches a supported version, then that version will be used to process the request.

If the version header is supplied and does not match a supported version, then the response will have status 501.

```
% curl -H "X-API-Version: 1.0.0" https://app.liquidplanner.com/api/account
{
  "message": "The version you requested is not implemented.",
  "error":   "NotImplemented",
  "type":    "Error"
}
Status: 501
```

Data Formats

JSON Responses

Responses to successful requests have a response body consisting of a JSON hash (for single records) or a JSON array of JSON hashes (for multiple records).

Date/Time Strings

Date/time representations should conform to ISO 8601

This format is:

```
[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss][Z]
```

where *Z* is a literal 'Z' for GMT, or (+/-)hh:mm for an offset from GMT.

HTTP Status Codes

Check the HTTP status code of responses from the API to determine whether your request was processed successfully.

Success Codes

200 OK
: successful read, update, or delete

201 Created
: successful create

Error or Exception Codes

400 Bad Request
: unrecognized or invalid request URL

401 Unauthorized
: credentials not provided or are incorrect

422 Unprocessable Entity
: unsuccessful create, update, or delete; syntactic or validation error

404 Not Found
: record does not exist, or you do not have permission to access it

500 Internal Server Error
: unexpected error; you've likely found a bug in LiquidPlanner

501 Not Implemented
: you asked for an API version that is not implemented (either because it does not exist, or it has been deprecated)

503 Service Unavailable
: API is disabled globally, or your account is throttled; check the error in the response body

Error Responses

Response Body

When returning an error, the response body contains details of the error represented as a JSON hash.

For example, requesting the invalid "/bad_url" results in:

```
% curl https://app.liquidplanner.com/api/bad_url
 {
   "type":    "Error",
   "message": "Probably a typo (or junk) in your request: /api/bad_url",
   "error":   "BadRequest"
 }
 Status: 400
```

Hash Keys

The hash keys in an error response include:

type
: the literal string "Error"

error
: specific kind of error, e.g. "NotFound" or "BadRequest"

message
: descriptive or explanatory message