|  |  |
| --- | --- |
| WinChatty v2 API | © 2013 Brian Luft. Updated: 2013-12-29 5:03 PM [GitHub](https://github.com/electroly/winchatty-server) • [Shackmessage](https://www.shacknews.com/messages?method=compose&to=electroly) • [Chatty Discussion](http://www.shacknews.com/article/69055) |

Contents

[Introduction 1](#_Toc376099918)

[Protocols 1](#_Toc376099919)

[Data Types 1](#_Toc376099920)

[Error Handling 1](#_Toc376099921)

[Client Implementation Guidelines <TODO> 1](#_Toc376099922)

[Threads 1](#_Toc376099923)

[GET /v2/getBumpedThreadIds 1](#_Toc376099924)

[GET /v2/getThread 1](#_Toc376099925)

[GET /v2/getThreadPostIds 1](#_Toc376099926)

[GET /v2/getSubthread <TODO> 1](#_Toc376099927)

[GET /v2/getSubthreadPostIds <TODO> 1](#_Toc376099928)

[Posts 1](#_Toc376099929)

[GET /v2/waitForNewPost 1](#_Toc376099930)

[GET /v2/getNewestPostInfo 1](#_Toc376099931)

[GET /v2/getPost 1](#_Toc376099932)

[GET /v2/getPostRange 1](#_Toc376099933)

[GET /v2/getParentId 1](#_Toc376099934)

[POST /v2/postComment 1](#_Toc376099935)

[GET /v2/search 1](#_Toc376099936)

[Events 1](#_Toc376099937)

[GET /v2/waitForEvent <TODO> 1](#_Toc376099938)

[GET /v2/getNewestEventId <TODO> 1](#_Toc376099939)

[GET /v2/getNewerEvents <TODO> 1](#_Toc376099940)

[Users 1](#_Toc376099941)

[POST /v2/verifyCredentials 1](#_Toc376099942)

[GET /v2/getUserRegistrationDate <TODO> 1](#_Toc376099943)

[Shackmessages 1](#_Toc376099944)

[POST /v2/getMessages 1](#_Toc376099945)

[POST /v2/sendMessage 1](#_Toc376099946)

[POST /v2/markMessageRead 1](#_Toc376099947)

[POST /v2/deleteMessage 1](#_Toc376099948)

[Client Data 1](#_Toc376099949)

[POST /v2/clientData/getClientSessionToken 1](#_Toc376099950)

[POST /v2/clientData/getCategoryFilters 1](#_Toc376099951)

[POST /v2/clientData/setCategoryFilters 1](#_Toc376099952)

[POST /v2/clientData/getMarkedPosts 1](#_Toc376099953)

[POST /v2/clientData/clearMarkedPosts 1](#_Toc376099954)

[POST /v2/clientData/markPost 1](#_Toc376099955)

[POST /v2/clientData/getPrivateData 1](#_Toc376099956)

[POST /v2/clientData/setPrivateData 1](#_Toc376099957)

[Appendix: Database Structure 1](#_Toc376099958)

# Introduction

I now have a complete chatty database that is continuously kept up to date as new posts are added. This means an API can be built that hits this database directly rather than accessing shacknews.com for every request. The existing LC.app API at [winchatty.com/chatty](http://www.winchatty.com/chatty) will be updated to hit the new database, but with direct access I can provide a much better API. The existing API was constrained by needing to scrape its data from shacknews.com pages.

It should be possible to write a full client application using only the v2 API. However, version 1 (the original LC.app / stonedonkey API) is not obsolete. The existing API calls will be plugged into the new chatty database <TODO>. So there is no problem with existing clients continuing to use the v1 API as convenience dictates.

Version 1 root: <https://winchatty.com/chatty/>  
Version 2 root: <https://winchatty.com/v2/>

This is a work in progress. Please shackmessage your comments to me. It's easy to add new API calls, so send in your ideas.

## Protocols

The v2 API operates via HTTP or HTTPS. The client may choose either one, but there's little reason to use HTTP. I recommend using HTTPS for everything. All responses are JSON.

If possible, client applications should be configured to verify SSL certificates:

* **libcurl (PHP)**  
  Use curl\_setopt() to set CURLOPT\_SSL\_VERIFYPEER to TRUE and CURLOPT\_SSL\_VERIFYHOST to 2.
* **WinInet (C++)**  
  When calling HttpOpenRequest(), use the flag INTERNET\_FLAG\_SECURE.   
  Do *not* use INTERNET\_FLAG\_IGNORE\_CERT\_CN\_INVALID or INTERNET\_FLAG\_IGNORE\_CERT\_DATE\_INVALID.
* **.NET**  
  WebClient automatically verifies SSL certificates. WebException is thrown if the certificate is invalid.
* **iOS / OS X**Based on some quick Google searches, I think NSURLConnection verifies SSL certificates by default and blows up in some way if the certificate is invalid.

The v2 API does not use cookies (likewise it does not use PHP sessions). When stateful interactions are required, the client manually requests and then keeps track of a session token which is passed with every request. At the moment, this only applies to the /v2/clientData/ calls. The majority of API calls are stateless.

The v2 API does not use HTTP authentication. Usernames and passwords, when applicable, are passed via POST arguments. Again, it is highly recommended that HTTPS be used so that usernames and passwords are not transmitted in plain text.

## Data Types

In order to precisely define the accepted inputs (query parameters) and the expected outputs (JSON) of the v2 API methods, the following data type shorthands are defined. Most types appear in both query parameters and JSON responses, but a few only appear in JSON responses.

|  |  |
| --- | --- |
| **Request and response types** | |
| [INT] | Integer |
| [BIT] | true or false |
| [STR] | String |
| [DAT] | Combined date and time, represented as a strict subset of [RFC 3339](http://tools.ietf.org/html/rfc3339), which is itself a strict subset of [ISO 8601](http://en.wikipedia.org/wiki/ISO_8601#Combined_date_and_time_representations). Example: "2013-12-01T19:39:00Z"  Dates passed in as query arguments are parsed using PHP's strtotime() which is very lenient. It will accept just about anything that looks remotely like a date. If no time zone is specified, then the local time in Milwaukee is assumed (either CST or CDT). It is recommended that input dates follow the strict format specified below (including the explicit use of UTC).  Output dates follow a strict format which avoids most of the complexity in ISO 8601 and even the simplified RFC 3339, permitting simple construction of a parser. Dates in JSON responses will always be formatted exactly like this: "2013-12-01T19:39:00Z". The following rules for output dates go beyond what ISO 8601 mandates (and beyond what the API requires for input dates).   * The ISO 8601 extended format (with hyphens in the date and colons in the time) will *always* be used. * The date will *always* include the day. * The time will *always* include the second. * The time will *never* include the millisecond. * The time zone designator will *always* be present. * The "Z" form of the time zone designator (indicating UTC) will *always* be used.   Thus the date string will always be exactly 20 characters long, and you may hardcode character offsets when writing your parser.  **Tip:** Make sure to convert all [DAT] values to the user's local time zone before displaying! |
| [MOD] | Moderation flag enum. One of the following strings:   * "ontopic" * "nws" * "stupid" * "political" * "tangent" * "informative" |
| [MBX] | Mailbox enum. One of the following strings:   * "inbox" * "sent" |
| [MPT] | Marked post type enum. One of the following strings:   * "unmarked" * "pinned" * "collapsed" |
| **Response-only types** | |
| [MODN] | Moderation flag enum, including "nuked". One of the following strings:   * "ontopic" * "nws" * "stupid" * "political" * "tangent" * "informative" * "nuked" |
| [POST] | A single post.  {  "id": [INT],  "threadId": [INT],  "parentId": [INT],  "storyId": [INT], <TODO>  "author": [STR],  "category": [MOD],  "date": [DAT],  "body": [STR]  } |
| [POSTS] | A list of posts.  [  [POST\*]  ] |
| [EVENT] | A single event of any type.  {  "id": [INT],  "type": [E\_TYPE],  "data": [E\_DATA] // check "type" to determine the concrete data type } |
| [EVENTS] | A list of events.  [  [EVENT\*] ] |
| [E\_TYPE] | Event action type enum. One of the following strings:   * "newPost" // data will be [E\_NEWP] * "categoryChange" // data will be [E\_CATC] * "serverMessage" // data will be [E\_SMSG] |
| [E\_DATA] | Event-specific data. Abstract base type which may be any one of the following concrete types:   * [E\_NEWP] * [E\_CATC] * [E\_SMSG] |
| [E\_NEWP] | New post event data.  {  "postId": [INT],  "post": [POST?] // not present if the post was later nuked } |
| [E\_CATC] | Category change event data.  {  "postId": [INT],  "oldCategory": [MODN?], // not present for stalled posts  "newCategory": [MODN]  } |
| [E\_SMSG] | Server message event data.  {  "message": [STR] } |

The following suffixes may appear on any of the data types above:

* The suffix + indicates a list of one or more, separated by comma.
* The suffix ? indicates that the argument may be omitted or empty.
* The suffix \* is the combinaton of + and ? (i.e. a list of zero or more).
* A comma and a number indicates the maximum value for integer arguments, the maximum count for list arguments, and the maximum length for string arguments.

## Error Handling

If an API call results in an error, it is returned in the following JSON structure.

{  
 "error": true,  
 "code": [STR],  
 "message": [STR]  
}

The documentation for each API call lists which error codes are possible. The following two error codes are possible on any API call, and are thus not listed on each individual call. In both cases it is recommended that the client simply display the error message and then cancel whatever operation caused it.

|  |  |
| --- | --- |
| ERR\_SERVER | Unexpected error. Could be a communications failure, Shacknews outage, PHP exception, etc. The client did not do anything wrong. |
| ERR\_ARGUMENT | Invalid argument. The client passed an argument value that violates a documented constraint. The client contains a bug. |

## Client Implementation Guidelines <TODO>

TODO: Fill this section with tips on implementing clients, based on our experience implementing the API in Lamp.

* Call getBumpedThreadIds to get the initial state of the chatty.
* Call getThread to initially get each of those threads.
* Start two loops:
  + waitForNewPost in a tight loop
  + getNewerCategoryChanges in a 1-minute loop

# Threads

These API calls relate to the chatty itself. These are the core of the v2 API.

## GET /v2/getBumpedThreadIds

Gets the list of recently bumped threads, starting with the most recently bumped. The root ID of each thread is returned, along with the date of the original post.

**Parameters**

count=[INT?]

The number of threads to return. If not specified, then all active (not expired) threads are returned.

expiration=[INT?,36]

The number of hours to keep threads around in this list. If not provided, then the default of 18 (to match Shacknews) is used. The maximum is 36 hours.

storyId=[INT?] <TODO>

Restrict the threads returned to a particular story (OldShack chatty or Weekend Confirmed article). If not provided, then normal chatty threads (story ID of 0) are returned.

**Response**

{  
 "threads":  
 [  
 {  
 "threadId": [INT],  
 "date": [DAT]  
 },  
 ... // one for each thread  
 ]  
}

**Examples**

<http://winchatty.com/v2/getBumpedThreadIds>  
<http://winchatty.com/v2/getBumpedThreadIds?expiration=24>  
<http://winchatty.com/v2/getBumpedThreadIds?count=30>  
<http://winchatty.com/v2/getBumpedThreadIds?count=30&expiration=24>

## GET /v2/getThread

Gets all of the posts in one or more threads. If an invalid ID is passed (or if the ID of a nuked post is passed), then that thread will be silently omitted from the resulting list of threads.

**Parameters**

id=[INT+,50]

One or more IDs. May be any post in the thread, not just the OP.

**Response**

{  
 "threads":  
 [  
 {  
 "threadId": [INT],  
 "posts": [POSTS]  
 },  
 ... // one for each thread  
 ]  
}

**Examples**

<http://winchatty.com/v2/getThread?id=31162211,31162001>

## GET /v2/getThreadPostIds

Gets the ID of each post in one or more threads. If an invalid ID is passed (or if the ID of a nuked post is passed), then that thread will be silently omitted from the resulting list of threads.

**Parameters**

id=[INT+,50]

One or more IDs. May be any post in the thread, not just the OP.

**Response**

{  
 "threads":  
 [  
 {  
 "threadId": [INT],  
 "postIds":  
 [  
 [INT],  
 ... // one for each post in the thread  
 ]  
 },  
 ... // one for each thread  
 ]  
}

**Examples**

<http://winchatty.com/v2/getThreadPostIds?id=31162211,31162001>

## GET /v2/getSubthread <TODO>

Gets all of the posts in one or more subthreads. A subthread is a post (which may or may not be a thread OP) and its descendants. If an invalid ID is passed (or if the ID of a nuked post is passed), then that thread will be silently omitted from the resulting list of subthreads.

**Parameters**

id=[INT+,50]

One or more IDs. The subthreads rooted at these IDs are returned.

**Response**

{  
 "subthreads":  
 [  
 {  
 "subthreadId": [INT],  
 "posts": [POSTS]  
 },  
 ... // one for each subthread  
 ]  
}

**Examples**

TODO

## GET /v2/getSubthreadPostIds <TODO>

Gets the ID of each post in one or more subthreads. A subthread is a post (which may or may not be a thread OP) and its descendants. If an invalid ID is passed (or if the ID of a nuked post is passed), then that thread will be silently omitted from the resulting list of subthreads.

**Parameters**

id=[INT+,50]

One or more IDs. The subthreads rooted at these IDs are returned.

**Response**

{  
 "subthreads":  
 [  
 {  
 "subthreadId": [INT],  
 "postIds":  
 [  
 [INT],  
 ... // one for each post in the subthread  
 ]  
 },  
 ... // one for each subthread  
 ]  
}

**Examples**

TODO

# Posts

These API calls relate to the chatty itself. These are the core of the v2 API.

## GET /v2/waitForNewPost

Waits until a new post appears, and then returns the newest post ID. If writing a full-blown client, you should use waitForEvent instead, as this covers all possible changes, not just new posts.

**Parameters**

lastId=[INT?]

Wait until any post newer than this ID appears. If a newer post already exists, then the request returns immediately without waiting. If not provided, then it will always wait until a new post appears.

**Response**

{  
 "id": [INT]   
}

**Examples**

<http://winchatty.com/v2/waitForNewPost>   
<http://winchatty.com/v2/waitForNewPost?lastId=31224456>

## GET /v2/getNewestPostInfo

Gets the ID and date of the most recent post in the database.

**Parameters**

None.

**Response**

{  
 "id": [INT],  
 "date": [DAT]  
}

**Examples**

<http://winchatty.com/v2/getNewestPostInfo>

## GET /v2/getPost

Gets one or more individual posts, specified by ID.

**Parameters**

id=[INT+,50]

The post IDs to retrieve.

**Response**

{  
 "posts": [POSTS]  
}

**Examples**

<http://winchatty.com/v2/getPost?id=31161163,31161164,31162308>

## GET /v2/getPostRange

Gets a consecutive range of posts. If any posts in the range do not exist (i.e. nuked, or hasn't been posted yet), then they are silently omitted from the list of posts in the response, rather than raising an error. The nuked posts are not counted against the number of posts requested by the count argument.

**Parameters**

startId=[INT]

The starting ID. This ID is included in the range.

count=[INT,1000]

Maximum number of posts to return, including startId.

reverse=[BIT?]

If true, then post IDs ≤ startId are retrieved. If not specified, or false, then post IDs ≥ startId are retrieved.

**Response**

{  
 "posts": [POSTS]  
 }

**Examples**

<http://winchatty.com/v2/getPostRange?startId=31158593&count=100>  
<http://winchatty.com/v2/getPostRange?startId=31158593&count=100&reverse=true>

## GET /v2/getParentId

Gets the parent IDs for one or more posts. If a post does not exist, then it is silently omitted from the list of relationships in the response, rather than raising an error. If a post is the OP of a thread, then the ID 0 is returned.

**Parameters**

id=[INT+,50]

List of post IDs. The parent ID of each one will be returned.

**Response**

{  
 "relationships":  
 [  
 {  
 "childId": [INT],  
 "parentId": [INT]  
 },  
 ... // one for each ID  
 ]  
}

**Examples**

<http://winchatty.com/v2/getParentId?id=3,31162309,31162346>

## POST /v2/postComment

Posts a new comment.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

parentId=[INT]

The ID of the post we're replying to, or 0 for a new thread.

text=[STR]

The body of the post.

newThreadStoryId=[INT?] <TODO>

The story ID to use when posting a new thread (*only* applicable for new threads, not new replies). If not specified, then 0 is the default, indicating that a regular chatty thread is desired. This allows threads to be created in Weekend Confirmed articles.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_LOGIN  
ERR\_POST\_RATE\_LIMIT  
ERR\_BANNED

**Example**

<https://winchatty.com/v2/postComment.tester>

## GET /v2/search

Performs a comment search.

**Parameters**

terms=[STR?]

Search terms.

author=[STR?]

Author.

parentAuthor=[STR?]

Parent author.

category=[MOD?]

Moderation flag.

offset=[INT?]

Number of results to skip. 0 is the default, which gets the first page of results.

limit=[INT?,500]

Maximum number of results to return. 35 is the default. Larger limits may take a long time to retrieve.

oldestFirst=[BIT?]

Whether to get results oldest first. Default: false.

**Response**

{  
 "posts": [POSTS]  
}

**Examples**

<http://winchatty.com/v2/search?terms=xbone>  
<http://winchatty.com/v2/search?author=electroly>   
<http://winchatty.com/v2/search?parentAuthor=electroly>  
<http://winchatty.com/v2/search?category=nws>  
<http://winchatty.com/v2/search?category=nws&oldestFirst=true>

# Events

Events allow the server to inform the client of any changes that are made, which the client would need to know to keep its local copy of the chatty up to date. The following list describes all of the event types:

* "newPost" – A new post has been added.
* "categoryChange" – The category of an existing post has been modified.
* "serverMessage" – The server administrator wants to display a message to all connected users.

The category change event encompasses the following three things that may happen to a post after it is initially made:

* The post may be nuked (removed from the chatty).
* If the post was previously nuked, then it may be unnuked (reinstated in the chatty).
* The post may be flagged with a moderation category like "informative".

All three events are considered a change to the post's category. To make this work, the standard set of categories (ontopic, nws, stupid, political, tangent, informative) is augmented with the special flag "nuked". This gives us a nice way to represent nukes, unnukes, and flags the same way: as a change to the post category

The nuke and unnuke operations may also happen due to Shacknews idiosyncrasies. For the purposes of the v2 API, a "nuked" post is simply a post ID that does not exist. It cannot distinguish between a post that once existed but was nuked by a moderator, and a post that never showed up due to a bug. In some situations I have witnessed, Shacknews will consume a post ID, but no post ever shows up, not even for a brief moment. It's as if the post was instantly nuked. The indexer will see this as being nuked right from the get-go, and will register it as a nuked post. It will log a "nuked" change. If the post finally appears some time later (which happens sometimes; the indexer will recheck a nuked post periodically afterwards), then the indexer will see it and then log an "unnuke" change, since the post didn't exist before and now it exists.

## GET /v2/waitForEvent <TODO>

Waits until a new event occurs, and then returns the information about any events that occurred since the last event seen by the client (as specified in the lastEventId argument). This is the primary method by which the client's local copy of the world is kept up-to-date. The client should process all events in sequential (by numeric ID) order.

A maximum of 10 events are returned. If more are required (e.g. because your lastEventId is stale), then use [getNewerEvents](#_GET_/v2/getNewerEvents_<TODO>). Or, if you're very far out of date, consider throwing out your world and starting over using the [getBumpedThreadIds](#_GET_/v2/getBumpedThreadIds) and [getThread](#_GET_/v2/getThread) calls.

**Parameters**

lastEventId=[INT?]

Wait until any event newer than this ID appears. If a newer event already exists, then the request returns immediately without waiting. If not provided, then it will take the current highest event ID and then wait for one higher than that (thus, if lastEventId is not provided, it will always block rather than returning immediately).

**Response**

{  
 "lastEventId": [INT], // new lastEventId to be used in your next loop  
 "events": [EVENTS]   
}

**Examples**

TODO

## GET /v2/getNewestEventId <TODO>

Gets the most recent event in the database.

**Parameters**

None.

**Response**

{  
 "eventId": [INT]  
}

**Examples**

TODO

## GET /v2/getNewerEvents <TODO>

Gets all events newer than a certain event ID. A maximum of 500 events are returned. If you hit this maximum, consider throwing out your world and starting over using the [getBumpedThreadIds](#_GET_/v2/getBumpedThreadIds) and [getThread](#_GET_/v2/getThread) calls rather than trying to catch up through a giant list of events.

**Parameters**

lastEventId=[INT]

Returns all events with an ID higher than lastEventId (exclusive).

**Response**

{  
 "events": [EVENTS]  
}

**Examples**

TODO

# Users

These API calls pertain to Shacknews user accounts.

## POST /v2/verifyCredentials

Checks the validity of the given username and password.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

**Response**

{  
 "isValid": [BIT]  
}

**Examples**

<https://winchatty.com/v2/verifyCredentials.tester>

## GET /v2/getUserRegistrationDate <TODO>

Gets the registration date for one or more users.

**Parameters**

username=[STR+,50]

List of Shacknews usernames.

**Response**

{  
 "users":  
 [  
 {  
 "username": [STR],  
 "date": [DAT]  
 },  
 ... // one for each user  
 ]  
}

**Errors**

TODO

# Shackmessages

The Shackmessage calls go directly to shacknews.com, as they did in the v1 API. Shackmessages are not stored in the WinChatty database to ensure user privacy is maintained.

## POST /v2/getMessages

Gets a page of messages in the user’s inbox or sent mailbox.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

folder=[MBX]

The mailbox folder.

page=[INT]

1-based page number.

**Response**

{  
 "page": [INT],  
 "totalPages": [INT],  
 "totalMessages": [INT],  
 "messages":  
 [  
 {  
 "id": [INT],  
 "from": [STR],  
 "to": [STR],  
 "subject": [STR],  
 "date": [DAT],  
 "body": [STR],  
 "unread": [BIT]  
 },  
 ... // one for each message  
 ]  
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<https://winchatty.com/v2/getMessages.tester>

## POST /v2/sendMessage

Sends a Shackmessage.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

to=[STR]

Message recipient's username.

subject=[STR]

Subject line.

body=[STR]

Post body.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<https://winchatty.com/v2/sendMessage.tester>

## POST /v2/markMessageRead

Marks a message as read. If the message does not exist, then the method returns successfully without doing anything.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

messageId=[INT]

Message ID.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<https://winchatty.com/v2/markMessageRead.tester>

## POST /v2/deleteMessage

Deletes a message. If the message does not exist, then the method returns successfully without doing anything.

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

messageId=[INT]

Message ID.

folder=[MBX]

"inbox" or "sent"

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<https://winchatty.com/v2/deleteMessage.tester>

# Client Data

The v2 API supports server storage ("cloud synchronization") of client data (primarily user preferences, but it's really just a general purpose store for the client's discretionary use). There are two types of client data associated with each user:

* *Shared* client data is common to all clients. For instance, the user's post filters (nws, political, etc.) are shared because every client supports this filtering feature. These clients can support cloud synchronization of this preference by reading and writing this shared data. All the shared data is available via formalized API methods with well-defined types and formats.
* *Private* client data is different for each client. Here the client can store its own preferences and data which necessarily cannot be shared with other clients. For instance, window positions, client-specific feature preferences, etc. This data is available via generic string read/write methods. It is recommended that you Base64-encode your data before passing it to this API.

Access to client data requires identifying your client application to the API. This is done by choosing a unique identification code to represent your client, which can (and should) simply be your application name. This ensures that you will see your own client's private data, and not some other client's data. You do not need to register this identification code ahead of time; simply call getClientSessionToken with your chosen code to get started. We'll operate by the honor system; choose something that's obviously unique and don't interfere with other clients. Don't be a dick.

## POST /v2/clientData/getClientSessionToken

Verifies the specified credentials and returns a token that can be used with future Client Data API calls. This allows the API to quickly check whether the caller has the rights to access a user's data without needing to hit shacknews.com each time (a 1-2 second operation). It will hit shacknews.com once for this call, and then not again for future API calls.

The token is valid until the expiration date and time specified in the response. If you call this method again before the expiration is up, then the expiration of the existing token is extended and the existing token is returned (as opposed to creating a new token).

**Parameters**

username=[STR]

Shacknews username.

password=[STR]

Shacknews password.

client=[STR]

Client identification code.

version=[STR]

Client version number. You may specify your version number in any format you choose.

**Response**

{  
 "clientSessionToken": [STR],  
 "expiration": [DAT]  
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<https://winchatty.com/v2/clientData/getClientSessionToken.tester>

## POST /v2/clientData/getCategoryFilters

Gets the user's moderation flag filters. A value of true indicates that posts in that category are shown.

**Parameters**

clientSessionToken=[STR]

Client session token.

**Response**

{  
 "filters":  
 {  
 "nws": [BIT],  
 "stupid": [BIT],  
 "political": [BIT],  
 "tangent": [BIT],  
 "informative": [BIT] <TODO>  
 }  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/getCategoryFilters.tester>

## POST /v2/clientData/setCategoryFilters

Sets the user's moderation flag filters. A value of true indicates that posts in that category are shown.

**Parameters**

clientSessionToken=[STR]

Client session token.

nws=[BIT]

Not work safe filter.

stupid=[BIT]

Stupid filter.

political=[BIT]

Political/religious filter.

tangent=[BIT]

Tangent filter.

informative=[BIT] <TODO>

Informative filter.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/setCategoryFilters.tester>

## POST /v2/clientData/getMarkedPosts

Gets all the user's marked posts (pinned or collapsed).

**Parameters**

clientSessionToken=[STR]

Client session token.

**Response**

{  
 "markedPosts":  
 [  
 {  
 id: [INT],  
 type: [MPT]  
 },  
 ... // one for each marked thread  
 ]  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/getMarkedPosts.tester>

## POST /v2/clientData/clearMarkedPosts

Clears the user's marked posts.

**Parameters**

clientSessionToken=[STR]

Client session token.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/clearMarkedPosts.tester>

## POST /v2/clientData/markPost

Marks a post as unmarked, pinned, or collapsed. The default for a regular post is unmarked.

**Parameters**

clientSessionToken=[STR]

Client session token.

postId=[INT]

Post ID.

type=[MPT]

Mark type.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_TOKEN  
ERR\_POST\_DOES\_NOT\_EXIST

**Examples**

<https://winchatty.com/v2/clientData/markPost.tester>

## POST /v2/clientData/getPrivateData

Gets the private client data for the specified user.

**Parameters**

clientSessionToken=[STR]

Client session token.

**Response**

{  
 "data": [STR]  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/getPrivateData.tester>

## POST /v2/clientData/setPrivateData

Sets the private client data for the specified user.

**Parameters**

clientSessionToken=[STR]

Client session token.

data=[STR]

Private client data. I recommend Base64-encoding this data.

**Response**

{  
 "result": "success"  
}

**Errors**

ERR\_INVALID\_TOKEN

**Examples**

<https://winchatty.com/v2/clientData/setPrivateData.tester>

# Appendix: Database Structure

All of the chatty data is stored in a PostgreSQL database. Searching is done using PostgreSQL's built-in text search functionality.

Threads and posts are stored in the following structure:



The following tables comprise the database schema.

|  |  |  |  |
| --- | --- | --- | --- |
| **indexer** | | | State information for the post indexer. |
| next\_low\_id | INTEGER | NOT NULL | The next oldest ID for the indexer to check. |
| next\_high\_id | INTEGER | NOT NULL | The next newest ID for the indexer to check. |
|  | | |  |
| **thread** | | | A comment thread. |
| id | INTEGER | PRIMARY KEY | The ID of the root post of the thread. |
| date | TIMESTAMP | NOT NULL | The post date of the root post. |
| bump\_date | TIMESTAMP | NOT NULL | The post date of the most recent post. |
|  | | |  |
| **post** | | | A single comment (root or reply) in a thread. |
| id | INTEGER | PRIMARY KEY | Post ID. |
| thread\_id | INTEGER | NOT NULL REFERENCES thread ON DELETE CASCADE | The ID of the root post of this thread. |
| parent\_id | INTEGER | NULL | The ID of the parent post, or 0 if this is the root. |
| author | TEXT | NOT NULL | Author username. |
| category | INTEGER | NOT NULL | Moderation flag: 1 = ontopic 2 = nws 3 = stupid 4 = political 5 = tangent 6 = informative |
| date | TIMESTAMP | NOT NULL | Post date. |
| body | TEXT | NOT NULL | Post body (including Shacktags in HTML). |
| author\_c | TEXT | NOT NULL | Preprocessed author, for searching. |
| body\_c | TEXT | NOT NULL | Preprocessed body, for searching. |
|  | | |  |
| **post\_index** | | | Contains the tokenized/stemmed body for searching. |
| id | INTEGER | NOT NULL REFERENCES post ON DELETE CASCADE | Post ID. |
| body\_c\_ts | tsvector | NOT NULL | Tokenized and stemmed body. |
| PRIMARY KEY (id) | | |  |
|  | | |  |
| **nuked\_post** | | | A nuked or missing post. |
| id | INTEGER | PRIMARY KEY | Post ID. |
| reattempts | INTEGER | NOT NULL | Number of times the indexer retried. |
| last\_date | TIMESTAMP | NOT NULL | Last time the indexer retried. |
| error | TEXT | NOT NULL | Error message the indexer received. |
|  | | |  |
| **post\_edit** | | | An audit log for a post modification. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| post\_id | INTEGER | NOT NULL | Post ID. May or may not be nuked (see category). |
| category | INTEGER | NOT NULL | Same as post.category, plus: 7 = nuked. |
| date | TIMESTAMP | NOT NULL | Date at which the post was modified. |
|  | | |  |
| **shacker** | | | A Shacknews user. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| username | TEXT | NOT NULL UNIQUE | Lowercase username. |
| signup\_date | TIMESTAMP | NULL | Signup date (retrieved on demand). |
| filter\_nws | BOOLEAN | NOT NULL | Client shared data: Show NWS posts? |
| filter\_stupid | BOOLEAN | NOT NULL | Client shared data: Show stupid posts? |
| filter\_political | BOOLEAN | NOT NULL | Client shared data: Show political posts? |
| filter\_tangent | BOOLEAN | NOT NULL | Client shared data: Show tangent posts? |
|  | | |  |
| **shacker\_marked\_post** | | | A pinned or collapsed thread. |
| shacker\_id | INTEGER | NOT NULL REFERENCES shacker ON DELETE CASCADE | Internal shacker ID. |
| post\_id | INTEGER | NOT NULL REFERENCES post ON DELETE CASCADE | Pinned post ID. |
| mark\_type | INTEGER | NOT NULL | 1 = pinned. 2 = collapsed. |
| PRIMARY KEY (shacker\_id, post\_id) | | |  |
|  | | |  |
| **private\_client\_data** | | | Private client-specific data. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| shacker\_id | INTEGER | NOT NULL REFERENCES shacker ON DELETE CASCADE | Internal shacker ID. |
| client\_code | TEXT | NOT NULL | Client code (selected by the client author). |
| data | TEXT | NULL | Data string (recommended to be base64-encoded). |
|  | | |  |
| **client\_session** | | | An active client session. |
| token | TEXT | PRIMARY KEY | Client token. |
| username | TEXT | NOT NULL | Username (not lowercased). |
| client\_code | TEXT | NOT NULL | Client code (selected by the client author). |
| client\_version | TEXT | NOT NULL | Client version (selected by the client author). |
| expire\_date | TIMESTAMP | NOT NULL | Session expiration date. |