|  |  |
| --- | --- |
| **WinChatty v2 API** | © 2014 Brian Luft. Updated: 2014-01-05 8:37 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](#_Toc376717581)

[Protocols 1](#_Toc376717582)

[Data Types 1](#_Toc376717583)

[Error Responses 1](#_Toc376717584)

[Client Implementation Guide 1](#_Toc376717585)

[Security and Privacy Disclosures 1](#_Toc376717586)

[Threads 1](#_Toc376717587)

[GET /v2/getChatty 1](#_Toc376717588)

[GET /v2/getThread 1](#_Toc376717589)

[GET /v2/getThreadPostIds 1](#_Toc376717590)

[GET /v2/getThreadPostCount 1](#_Toc376717591)

[GET /v2/getSubthread 1](#_Toc376717592)

[Posts 1](#_Toc376717593)

[GET /v2/getNewestPostInfo 1](#_Toc376717594)

[GET /v2/getPost 1](#_Toc376717595)

[GET /v2/getPostRange 1](#_Toc376717596)

[GET /v2/getParentId 1](#_Toc376717597)

[POST /v2/postComment 1](#_Toc376717598)

[GET /v2/search 1](#_Toc376717599)

[POST /v2/requestReindex 1](#_Toc376717600)

[Events 1](#_Toc376717601)

[GET /v2/getNewestEventId 1](#_Toc376717602)

[GET /v2/waitForEvent 1](#_Toc376717603)

[GET /v2/pollForEvent 1](#_Toc376717604)

[POST /v2/broadcastServerMessage 1](#_Toc376717605)

[Users 1](#_Toc376717606)

[POST /v2/verifyCredentials 1](#_Toc376717607)

[GET /v2/getUserRegistrationDate 1](#_Toc376717608)

[GET /v2/getAllUserRegistrationDates 1](#_Toc376717609)

[Shackmessages 1](#_Toc376717610)

[POST /v2/getMessages 1](#_Toc376717611)

[POST /v2/sendMessage 1](#_Toc376717612)

[POST /v2/markMessageRead 1](#_Toc376717613)

[POST /v2/deleteMessage 1](#_Toc376717614)

[Client Data 1](#_Toc376717615)

[POST /v2/clientData/getClientSessionToken 1](#_Toc376717616)

[POST /v2/clientData/getCategoryFilters 1](#_Toc376717617)

[POST /v2/clientData/setCategoryFilters 1](#_Toc376717618)

[POST /v2/clientData/getMarkedPosts 1](#_Toc376717619)

[POST /v2/clientData/clearMarkedPosts 1](#_Toc376717620)

[POST /v2/clientData/markPost 1](#_Toc376717621)

[POST /v2/clientData/getPrivateData 1](#_Toc376717622)

[POST /v2/clientData/setPrivateData 1](#_Toc376717623)

[Appendix: Database Structure 1](#_Toc376717624)

# 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) has been retrofitted to hit the new database, but with direct access I can provide a much cleaner 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. There is no problem with existing clients continuing to use the v1 API as convenience dictates. There are no plans to deprecate the v1 API.

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

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.

Client applications should be configured to use GZIP compression and verify SSL certificates. These are both very important, but GZIP compression especially so. On average GZIP cuts the size of responses down by 75%.

* **libcurl**Use [curl\_setopt()](http://www.php.net/curl_setopt) to set:   
  CURLOPT\_SSL\_VERIFYPEER = true   
  CURLOPT\_SSL\_VERIFYHOST = 2  
  CURLOPT\_ENCODING = "gzip"
* **WinInet**Call [HttpOpenRequest()](http://msdn.microsoft.com/en-us/library/windows/desktop/aa384233(v=vs.85).aspx) using the flag INTERNET\_FLAG\_SECURE. Then call [InternetSetOption()](http://msdn.microsoft.com/en-us/library/windows/desktop/aa385114(v=vs.85).aspx) using the flag INTERNET\_OPTION\_HTTP\_DECODING.
* **.NET**[WebClient](http://msdn.microsoft.com/en-us/library/system.net.webclient(v=vs.110).aspx) automatically verifies SSL certificates, but you must override [GetWebRequest()](http://msdn.microsoft.com/en-us/library/system.net.webclient.getwebrequest(v=vs.110).aspx) in order to support GZIP compression (see [this StackOverflow answer](http://stackoverflow.com/a/4914874) for the code). [WebException](http://msdn.microsoft.com/en-us/library/system.net.webexception(v=vs.110).aspx) is thrown if the certificate is invalid.
* **iOS / OS X**  
  Based on some quick Google searches, I think [NSURLConnection](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/Reference/Reference.html) verifies SSL certificates by default and blows up in some way if the certificate is invalid. I think you have to add the "Accept-Encoding" header in order to support GZIP compression (see [this StackOverflow answer](http://stackoverflow.com/a/2683986) for the code).

The v2 API does not use cookies; likewise it does not use PHP sessions. When stateful interactions are required, the client manually requests a session token which it passes as an argument with every request. At the moment, this only applies to the [/v2/clientData/](#_Client_Data) 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. It is highly recommended that HTTPS be used so that usernames and passwords are not transmitted in plain text.

Besides making the API easier to use, I believe the aforementioned restrictions on cookies and authentication are required when using [CORS](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing) to allow access from all domains. The API returns the header Access-Control-Allow-Origin: \*. The [W3C documentation for CORS](http://www.w3.org/TR/cors/#resource-requests) states:

* ["The string "\*" cannot be used for a resource that *supports credentials*."](http://www.w3.org/TR/cors/#resource-requests)
* ["A *supports credentials* flag that indicates whether the resource supports *user credentials* in the request."](http://www.w3.org/TR/cors/#supports-credentials)
* ["The term *user credentials* for the purposes of this specification means cookies, HTTP authentication, and client-side SSL certificates that would be sent based on the user agent's previous interactions with the origin."](http://www.w3.org/TR/cors/#user-credentials)

## 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] | Unsigned 32-bit decimal integer. No leading zeroes. |
| [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],  "author": [STR],  "category": [MOD],  "date": [DAT],  "body": [STR] } |
| [POSTS] | A list of posts.  [  [POST\*] ] |
| [EVENT] | A single event of any type.  {  "eventId": [INT],  "eventDate": [DAT],  "eventType": [E\_TYPE],  "eventData": [E\_DATA] // check "type" first } |
| [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 posts event data.  {  "postId": [INT],  "post": [POST?]  } |
| [E\_CATC] | Category change event data.  {  "postId": [INT],  "category": [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 Responses

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 Guide

These are general guidelines to follow when implementing a "full featured" client based on the v2 API. Feel free to pick and choose based on your client's unique needs. All of the API calls are designed to stand alone, as well as work in conjunction with the others.

At application startup:

* If your client provides cloud synchronization and the user has provided a username and password, call [/v2/clientData/getClientSessionToken](#_POST_/v2/clientData/getClientToken_). Use the other Client Data methods to retrieve the user's client options.
* Call [/v2/getNewestEventId](#_GET_/v2/getNewestEventId) and save the event ID. This ID will be continually updated as new events arrive.
* Call [/v2/getChatty](#_GET_/v2/getChatty) to bootstrap your local copy of the chatty, including all active threads.
* If your client shows lightning bolts for 10-year users, then call [/v2/getAllUserRegistrationDates](#_GET_/v2/getAllUserRegistrationDates) to bootstrap your list of registration dates. If you encounter a username that isn't in your list, then call [/v2/getUserRegistrationDate](#_GET_/v2/getUserRegistrationDate). For some usernames (specifically, usernames containing punctuation characters), this will fail. This is a shortcoming of Shacknews itself.

In a loop running until the application exits:

* *For desktop clients and other "unlimited energy/bandwidth/processor " scenarios:*  
  Call [/v2/waitForEvent](#_GET_/v2/waitForEvent), passing the last event ID (either from the original [/v2/getNewestEventId](#_GET_/v2/getNewestEventId) call, or the previous loop). This will block until an event is ready, so your loop does not need any artificial delays (unless you want to artificially limit the rate of events).
* *For mobile clients and other "limited energy/bandwidth/processor" scenarios:*  
  Call [/v2/pollForEvent](#_GET_/v2/pollForEvent), passing the last event ID (either from the original [/v2/getNewestEventId](#_GET_/v2/getNewestEventId) call, or the previous loop). This will always return immediately, but may return zero events. Then delay for length of time of your choosing (perhaps 1 minute) to allow your WiFi/3G/LTE radio to go idle.
* If ERR\_TOO\_MANY\_EVENTS is returned, then throw out your copy of the chatty and start over by calling [/v2/getNewestEventId](#_GET_/v2/getNewestEventId) and [/v2/getChatty](#_GET_/v2/getBumpedThreadIds). If the call fails with a different error, then display the error message and exit the loop rather than continuing to call it.

When your event loop retrieves a new event:

* For a new post, insert the post into your copy of the chatty.
* For a category change to "nuked", remove the post and all of its children from your copy of the chatty.
* For a category change to anything else, update the existing post in your copy of the chatty. If the post does not exist in your copy of the chatty, then it must have been previously nuked and now has been reinstated. Call [/v2/getSubthread](#_GET_/v2/getSubthread) to get the subthread rooted at this post, and insert all of the posts into your copy of the chatty.
* For a server message, show a message box to the user with the specified administrator message.

When the user changes a client option:

* If the username and password were changed, called [/v2/verifyCredentials](#_POST_/v2/verifyCredentials) to ensure the login information is valid. Then call [/v2/clientData/getClientSessionToken](#_POST_/v2/clientData/getClientToken_). Depending on your client's features, use the other Client Data methods to retrieve the user's client options.
* Use the Client Data methods to save the updated client options.

## Security and Privacy Disclosures

This section makes a serious effort to spell out the security and privacy implications of this API and its use in chatty clients. Primarily this relates to user's passwords (a security issue) and shackmessages (a privacy issue).

You have my word that the following statement is true:

* All of the source code running on [winchatty.com](http://winchatty.com/) matches the publicly available code on [GitHub](https://github.com/electroly/winchatty-server).

This is the only statement that you must accept on faith, because there is no way for you to verify it. In principle I could run any malicious code I wanted on the server, while committing a "cleaned-up" version to GitHub, and you'd never know it.

The following statements are also true, and you may verify them yourself by inspecting the code on GitHub.

* User passwords are never stored on disk in any form.
* Unencrypted user passwords are temporarily stored in the server's RAM while generating a response.
* Shackmessages are never stored on disk in any form.
* Unencrypted Shackmessages are temporarily stored in the server's RAM while generating a response.
* User passwords are never used for any purpose other than responding to explicit client requests.
* All API calls are logged, with originating IP address and GET arguments (standard Apache logs).

# Threads

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

## GET /v2/getChatty

Gets the list of recently bumped threads, starting with the most recently bumped. Only "active" threads (i.e. threads that have not expired) are included. Thus this essentially grabs the entire chatty, as seen from the Shacknews website. The full threads are returned. You should call this method to bootstrap your application's local copy of the chatty, and then use [waitForEvent](#_GET_/v2/waitForEvent) to keep it up to date.

**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.

**Response**

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

**Examples**

<http://winchatty.com/v2/getChatty>   
<http://winchatty.com/v2/getChatty?expiration=24>   
<http://winchatty.com/v2/getChatty?count=30>   
<http://winchatty.com/v2/getChatty?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/getThreadPostCount

Gets the number of posts in one or more threads, including the root post (i.e. the post count is always at least 1).

**Parameters**

id=[INT+,50]

One or more thread IDs. Must be the root post of the thread. If an ID is passed which isn't a thread root post, then it is silently omitted from the resulting list.

**Response**

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

**Examples**

<http://winchatty.com/v2/getThreadPostCount?id=31314961,31314962>

## GET /v2/getSubthread

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**

<http://winchatty.com/v2/getSubthread?id=31163042,31162321>

# Posts

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

## 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.

**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>

## POST /v2/requestReindex

Requests a reindex of a specified post. This is used when implementing moderator support. When the moderator nukes or flags a post, the client should call this method to notify the server about the change. This ensures that the database is immediately updated. Otherwise, it may take some time for the database to be updated.

**Parameters**

postId=[INT]

Post ID.

**Response**

{  
 "result": "success"  
}

**Example**

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

# 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.

## GET /v2/getNewestEventId

Gets the most recent event in the database.

**Parameters**

None.

**Response**

{  
 "eventId": [INT]  
}

**Examples**

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

## GET /v2/waitForEvent

Waits until a new event occurs, and then returns the information about all 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 100 events are returned. An error is returned if more than 100 events have occurred since your specified lastEventId. In that case, throw out your world and start over. This will be faster than trying to catch up with a massive list of individual updates.

**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.

**Response**

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

**Errors**

ERR\_TOO\_MANY\_EVENTS

**Examples**

<http://winchatty.com/v2/waitForEvent?lastEventId=40>

## GET /v2/pollForEvent

Returns the information about all events (if any) that occurred since the last event seen by the client (as specified in the lastEventId argument). This method is for use by clients in limited bandwidth or limited processor scenarios. It is expected that these clients would call this method around once per minute (the interval is up to the developer's discretion). Desktop clients (and phone clients who want a faster update rate at the expense of battery life) should use [/v2/waitForEvent](#_GET_/v2/waitForEvent). The client should process all events in sequential (by numeric ID) order.

A maximum of 100 events are returned. An error is returned if more than 100 events have occurred since your specified lastEventId. In that case, throw out your world and start over. This will be faster than trying to catch up with a massive list of individual updates.

**Parameters**

lastEventId=[INT]

Return any event newer than this ID.

**Response**

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

**Errors**

ERR\_TOO\_MANY\_EVENTS

**Examples**

<http://winchatty.com/v2/pollForEvent?lastEventId=40>

## POST /v2/broadcastServerMessage

Administrator-only method to broadcast a server message to all connected users.

**Parameters**

username=[STR]

Administrator username.

password=[STR]

Administrator password.

message=[STR]

Server message.

**Response**

{  
 "result": "success"   
}

**Errors**

ERR\_INVALID\_LOGIN

**Examples**

<http://winchatty.com/v2/broadcastServerMessage.tester>

# 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

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**

<http://winchatty.com/v2/getUserRegistrationDate?username=electroly,CRasterImage>

## GET /v2/getAllUserRegistrationDates

Gets a bulk dump of registration dates for all users.

**Parameters**

None.

**Response**

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

**Errors**

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

# 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](#_POST_/v2/clientData/getClientSessio) 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]  
 }  
}

**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]

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. |
|  | | |  |
| **event** | | | An audit log for all database updates. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| date | TIMESTAMP WITH TIME ZONE | NOT NULL | Event date. |
| type | TEXT | NOT NULL | Event type. |
| data | TEXT | NOT NULL | JSON event data. |
|  | | |  |
| **shacker** | | | A Shacknews user. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| username | TEXT | NOT NULL UNIQUE | Lowercase username. |
| signup\_date | TIMESTAMP WITH TIME ZONE | 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? |
| filter\_informative | BOOLEAN | NOT NULL | Client shared data: Show informative 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 WITH TIME ZONE | NOT NULL | Session expiration date. |
|  | | |  |
| **reindex\_request** | | | A pending request to reindex a post. |
| id | SERIAL | PRIMARY KEY | Internal ID. |
| post\_id | INTEGER | NOT NULL | Post ID. |