Package 'chat.api'

November 12, 2026

Type Package

Title Transport-Agnostic Chat Contract

Version 0.2.0

Date 2026-11-10

Description A transport-agnostic contract for chat-room connectivity: connect, poll, and send against one interface, with adapters for 'Matrix' <https://spec.matrix.org/>, 'Slack' <https://api.slack.com/>, 'Telegram' <https://core.telegram.org/bots/api>, and Internet Relay Chat (IRC). An in-memory adapter supports local testing. Capability flags describe support for threads, markup dialects, encryption, and per-message identity. Platform clients are supplied by optional packages; the core interface uses only base R.

License Apache License (>= 2)

Depends R (>= 4.0)

URL https://github.com/cornball-ai/chat.api

BugReports https://github.com/cornball-ai/chat.api/issues

Suggests httr, mx.api, mx.client (>= 0.3.0), mx.crypto (>= 0.3.0), slackr, telegram, tinytest

Encoding UTF-8

NeedsCompilation no

Author Troy Hernandez [aut, cre] (ORCID: <https://orcid.org/0009-0005-4248-604X>), cornball.ai [cph]

Maintainer Troy Hernandez <troy@cornball.ai>

Repository CRAN

Date/Publication 2026-11-11 23:20:09 UTC

Contents

chat.api-package . . . 3
chat_addressed . . . 3
chat_attachment . . . 4
chat_capabilities . . . 5
chat_channels . . . 6
chat_channel_create . . . 6
chat_channel_info . . . 7
chat_channel_invite . . . 8
chat_config . . . 9
chat_config_save . . . 9
chat_disconnect . . . 10
chat_download . . . 11
chat_edit . . . 12
chat_get_state . . . 13
chat_history . . . 14
chat_identity . . . 15
chat_invite . . . 16
chat_irc . . . 17
chat_join . . . 17
chat_leave . . . 18
chat_loopback . . . 19
chat_mark_read . . . 19
chat_matrix . . . 20
chat_matrix_config . . . 24
chat_matrix_configure . . . 25
chat_matrix_config_path . . . 26
chat_members . . . 27
chat_message . . . 27
chat_pending . . . 29
chat_poll . . . 30
chat_react . . . 31
chat_reaction . . . 32
chat_relogin . . . 33
chat_resolve . . . 33
chat_send . . . 34
chat_set_identity . . . 35
chat_set_state . . . 36
chat_slack . . . 37
chat_telegram . . . 39
chat_typing . . . 40
chat_whoami . . . 41

Index **42**

chat.api-package *Transport-agnostic chat connectivity*

Description

A common interface for chat messages, attachments, rooms, and identity. Use chat_loopback for local development, or connect through chat_matrix, chat_irc, chat_slack, or chat_telegram. Inspect chat_capabilities before using optional operations.

Examples

```
cl <- chat_loopback()
chat_send(cl, "general", "hello")
chat_poll(cl)$messages
```

chat_addressed *Does a message address this client?*

Description

Answers the question a bot in a room full of people has to ask before replying. Two signals feed it, and which ones exist is the adapter's business rather than the caller's:

Usage

```
chat_addressed(client, message, ...)
```

Arguments

client	A chat_client.
message	A chat_message.
...	Adapter-specific options.

Details

1. The message's declared mentions – Matrix m.mentions, a Slack user ref. Structured, unambiguous, and the only signal the default method reads.
2. The plain-text conventions of the transport. Matrix has @@bot and the full user id, Slack has <@U0123>, IRC has a leading nick:. These are the reason this is a verb and not a field: writing the Matrix form into a consumer is how that consumer ends up knowing it is talking to Matrix.

The default reads only the declared mentions, so an adapter that does not override it under-reports rather than over-reports. A bot that misses being addressed stays quiet; one that thinks it was addressed when it was not talks over people, and unprompted is worse than absent.

Value

TRUE or FALSE.

Examples

```
cl <- chat_loopback()
chat_send(cl, "general", "hi")
chat_addressed(cl, chat_poll(cl)$messages[[1L]])
```

`chat_attachment` *Construct a normalized attachment record*

Description

Inbound media on a chat_message: `chat_poll()` and `chat_history()` put these in the message's `attachments` on adapters whose `chat_capabilities()$attachments` is TRUE.

Usage

```
chat_attachment(
  id,
  name = NA_character_,
  mime = NA_character_,
  bytes = NA_integer_,
  url = NA_character_,
  path = NA_character_,
  sha256 = NA_character_,
  raw = NULL
)
```

Arguments

`id`	Adapter-native identifier for the content (a Matrix mxc URI, a Slack file id). The stable handle; everything else here is description.
`name`	Filename as the sender labeled it, or NA.
`mime`	MIME type, or NA when the transport does not say.
`bytes`	Size in bytes, or NA.
`url`	A fetchable location, or NA. It may require this client's credentials; a consumer must not assume it is public.
`path`	Local filesystem path when the content is already on disk, or NA.
`sha256`	Content hash, or NA. Adapters fill it only when the transport carries one (Matrix encrypted attachments do); they must not compute it speculatively, because NA meaning "unverified" is what tells a consumer that needs provenance to hash at ingest and record the result.
`raw`	The adapter's platform-native payload.

Value

A list with class `chat_attachment`.

Examples

```
chat_attachment("mxc://ex/abc", name = "plot.png", mime = "image/png")
```

`chat_capabilities` *Describe what a chat client's platform supports*

Description

Describe what a chat client's platform supports

Usage

```
chat_capabilities(client, ...)
```

Arguments

`client`	A `chat_client`.
`...`	Adapter-specific options.

Value

A list with at least: `threads` (can post into threads), `thread_replies` (thread replies come back out of `chat_poll`), `edits`, `reactions` (`chat_react` works), `reaction_events` (reactions come back out of `chat_poll`), `channel_info` (`chat_channel_info` works), `members` (`chat_members` works), `invites` (invitations come back out of `chat_poll`), `join` (`chat_join` works), `whoami` (`chat_whoami` works, and with it the default `chat_addressed`), `channel_create` (`chat_channel_create` works), `leave` (`chat_leave` works), `channel_invite` (`chat_channel_invite` works), `set_state` (durable channel state: both `chat_set_state` and `chat_get_state` work – a transport has the pair or neither), `files` (outbound: `chat_send(files =)` works), `attachments` (inbound: media comes back out of `chat_poll` as `chat_attachment` records), `typing`, `e2ee`, `avatar` (`chat_set_identity` can set the account's picture, not only its name), `identity_override` (logicals), `user_identity` (a send can authenticate as a real member of the platform rather than as the bot – Slack's `chat_send(as_user = TRUE)` with a user token; a property of the client instance's configuration there, and FALSE everywhere else), `markup_dialects` (character), `max_message_bytes` (integer or NA).

Sending and receiving get separate flags wherever a platform does one and not the other, which is why `threads` and `thread_replies` are two entries rather than one. Reactions split the same way: Slack can place one, and does not report anyone else's through the history endpoint this adapter polls, so a consumer reading a single flag would wait forever for events that never arrive.

Examples

```
caps <- chat_capabilities(chat_loopback())
caps$threads
caps$e2ee
```

chat_channels *List the channels this client is in*

Description

The state half of the contract. chat_poll answers "what changed since my cursor"; this and its siblings answer "what is true now", which is the question a process asks when it starts up with no useful cursor at all.

Usage

```
chat_channels(client, ...)
```

Arguments

client	A chat_client.
...	Adapter-specific options.

Value

Character vector of channel identifiers.

Examples

```
chat_channels(chat_loopback())
```

chat_channel_create *Create a channel*

Description

Capability-gated: check chat_capabilities()$channel_create. Bots open rooms about as often as they are invited to them; a contract without creation forces every such consumer below the seam, into adapter-native calls.

Usage

```
chat_channel_create(client, name, ...)
```

Arguments

client	A chat_client.
name	Character. Human-readable name for the new channel.
...	Adapter-specific options (topic, visibility, invitees).

Value

The new channel's identifier, invisibly. Everything else – sends, joins, membership – takes the identifier, not the name, so the return value is the point of the call.

Examples

```
cl <- chat_loopback()
id <- chat_channel_create(cl, "general")
chat_send(cl, id, "hello")
```

---

chat_channel_info *Describe a channel*

Description

Returns the channel's descriptive metadata: what it is called and what it is for.

Usage

```
chat_channel_info(client, channel, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
...	Adapter-specific options.

Details

Membership is deliberately not here, and chat_members is a separate verb. The two look like one lookup and are not: a name and a topic are two short strings that change rarely, while a member list is unbounded and changes constantly. Bundling them makes every read of a topic pay for a member list, which on a busy room is the expensive part – and a consumer caches the two on different schedules for exactly that reason.

A NULL field means the channel has no such thing: a Matrix room with no m.room.name really has no name. An adapter that cannot answer at all throws, so "cannot ask" and "asked, and there is none" stay distinguishable. Check chat_capabilities()$channel_info first.

Value

A list with id, name, and topic. id is the channel as the platform addresses it; name and topic are character or NULL.

Examples

```
## Not run:
# Requires a saved Matrix configuration and a joined room.
cl <- chat_matrix(app = "mybot")
chat_channel_info(cl, "#general:example.org")

## End(Not run)
```

`chat_channel_invite` *Invite a user to a channel*

Description

The other side of an invitation: `chat_poll` reports the ones this client receives, and this verb issues one. A bot that files rooms for people needs it about as often as it needs `chat_channel_create`, because a room nobody but the bot can enter does not exist for anyone else. Capability-gated: check `chat_capabilities()$channel_invite`.

Usage

```
chat_channel_invite(client, channel, user, ...)
```

Arguments

`client`	A `chat_client`.
`channel`	Channel/room identifier.
`user`	The invitee's platform identifier, such as a Matrix user id (`"@alice:example.org"`).
`...`	Adapter-specific options.

Details

The default method throws, on `chat_join`'s reasoning: an invitation that silently did not go out leaves the caller believing someone can see a room they cannot.

Value

The channel identifier, invisibly.

Examples

```
cl <- chat_loopback()
chat_channel_invite(cl, "general", "@alice:example.org")
```

chat_config *Construct a chat_config*

Description

Construct a chat_config

Usage

```
chat_config(x, app = NULL, path = NULL)
```

Arguments

x	A named list of configuration fields.
app	Application namespace the config belongs to, or NULL.
path	File the config was read from, or NULL for one that has never been written.

Value

A list with class chat_config.

Examples

```
chat_config(list(server = "https://ex.invalid", user = "bot"),
            app = "demo")
```

chat_config_save *Persist a configuration*

Description

Writes to the file the config came from, at mode 0600.

Usage

```
chat_config_save(config, app = NULL, path = NULL)
```

Arguments

config	A chat_config, or a plain list together with app/path.
app	Override the app namespace the config was loaded under.
path	Override the file to write.

Value

The config, invisibly.

Examples

```
if (requireNamespace("mx.client", quietly = TRUE)) {
  path <- tempfile(fileext = ".json")
  cfg <- chat_config(list(server = "https://matrix.example.org",
                          token = "example-token",
                          user_id = "@bot:example.org"))
  chat_config_save(cfg, path = path)
  file.exists(path)
  unlink(path)
}
```

`chat_disconnect` *Close a chat client's connection*

Description

The default method is a no-op: HTTP-poll transports have nothing to close. Persistent-socket transports (IRC) override it.

Usage

```
chat_disconnect(client, ...)
```

Arguments

`client`	A `chat_client`.
`...`	Adapter-specific options.

Value

TRUE, invisibly.

Examples

```
cl <- chat_loopback()
chat_disconnect(cl)
```

chat_download *Fetch an attachment's bytes to a local file*

Description

Inbound attachments name where their content lives, not what it is: chat_attachment's url is a platform handle that usually needs this client's credentials, so a consumer cannot simply download it. This is the verb that turns one into a file on disk.

Usage

```
chat_download(client, attachment, dest = NULL, ...)
```

Arguments

client	A chat_client.
attachment	A chat_attachment record, as carried on a chat_message's attachments.
dest	Destination path. NULL picks a temporary file, keeping the attachment's extension where it has one. The caller should remove temporary downloads with unlink() when finished.
...	Adapter-specific options.

Details

Capability-gated on chat_capabilities()$attachments, the same flag that says inbound media arrives at all. The default method throws, on chat_react's reasoning: a fetch that quietly did nothing leaves the caller pointing at a path with no bytes behind it.

An attachment already on disk (path set, as the loopback adapter records) is copied rather than fetched, so a consumer needs one code path for both.

Value

The destination path, invisibly.

Examples

```
cl <- chat_loopback()
src <- tempfile(fileext = ".txt")
writeLines("hello", src)
chat_send(cl, "general", "a file", files = src)
attachment <- chat_poll(cl)$messages[[1L]]$attachments[[1L]]
dest <- chat_download(cl, attachment)
readLines(dest)
unlink(c(src, dest))
```

chat_edit *Replace the text of a message already sent*

Description

What makes a progress message possible: post "working on it", then keep replacing it as the work happens, instead of narrating into the channel one message at a time.

Usage

```
chat_edit(
  client,
  channel,
  message_id,
  text,
  markup = c("plain", "markdown"),
  rich = NULL,
  kind = "message",
  ...
)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
message_id	The message to replace, as returned by chat_send.
text	The replacement text, in full. Not a delta: every platform that supports this takes the whole new body, and a contract that took a patch would have to reconstruct the old one to apply it.
markup	"plain" or "markdown", as chat_send.
rich	Adapter-native markup for the replacement, as chat_send: an HTML fragment where chat_capabilities()$rich_markup says so, ignored where it is empty. text is still the whole replacement on its own, for the same reason it is on a send.
kind	Message kind, as chat_send. Must match what the message was sent as: Matrix carries the msgtype inside the replacement content, so editing an m.notice without saying so turns it into an ordinary message.
...	Adapter-specific options.

Details

The default throws. An edit that silently does nothing leaves the old text on screen, and stale content is worse than a visible failure – the reader has no way to tell that what they are looking at is no longer true. Check chat_capabilities()$edits.

Value

The identifier of the event the edit created where the platform makes one (Matrix), or of the edited message where it does not (Slack), invisibly.

What a consumer must not assume

That the edit is what readers see. A client that does not implement edits shows the original and an "* edited" fallback beside it, and notifications almost always carry the text as first sent. So the first version has to stand on its own – "working on it" is a fine thing to be paged with, a half-finished sentence is not.

Examples

```
cl <- chat_loopback()
id <- chat_send(cl, "general", "Working on it")
chat_edit(cl, "general", id, "Finished")
chat_history(cl, "general")$messages
```

---

chat_get_state *Read durable typed state from a channel*

Description

The counterpart of chat_set_state, reading back what it wrote. Capability-gated on the same chat_capabilities()$set_state: a transport with durable channel state has both halves or neither.

Usage

```
chat_get_state(client, channel, type, state_key = "", ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
type	Character. Namespaced event type.
state_key	Character. Sub-key within the type, defaulting to the empty string.
...	Adapter-specific options.

Details

Absent state is NULL, not an error. "Nothing was ever written here" is an ordinary answer for a caller checking whether a marker is set, and one it should not have to wrap in a handler. A state store that cannot be reached at all still errors, because that is a different fact.

Value

The stored content as a named list, or NULL when no state is set for that type/state_key pair.

Examples

```
cl <- chat_loopback()
chat_get_state(cl, "general", "m.room.topic") # NULL until written
chat_set_state(cl, "general", "m.room.topic", list(topic = "Planning"))
chat_get_state(cl, "general", "m.room.topic")
```

---

chat_history *Read a channel's recent messages*

Description

Independent of the poll cursor: a restarted process uses this to recover the context it lost, and asking for it must not move the cursor or consume anything.

Usage

```
chat_history(client, channel, limit = 50L, cursor = NULL, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
limit	Maximum messages to return.
cursor	Opaque continuation token from a previous call's cursor, to read the page before it; NULL starts from the most recent.
...	Adapter-specific options.

Value

A list with messages (list of chat_message, oldest first) and cursor (opaque; pass it back to read further into the past, NULL when the channel has no more history).

The cursor is opaque, like chat_poll's

Not a message id. This started out taking one and it was wrong on the reference transport: Matrix's /messages takes a pagination token from a previous response, and handing it an event id does not page from that event – it fails, or worse, silently returns the wrong window. Slack pages by its own next_cursor. There is no id that means the same thing on both, so the contract does what it already does for chat_poll: the token is the adapter's, and a consumer only ever passes back what it was given.

Order

Chronological, oldest first, whatever the platform's native direction is. Matrix `dir = "b"` and Slack `conversations.history` both hand back newest-first and every consumer replaying history into a transcript has to flip it. One flip in the adapter beats one per consumer, and a consumer that gets it wrong produces a transcript that reads backwards without erroring.

Note that pages run backwards while each page runs forwards: call it twice and the second page's messages all precede the first page's. A consumer assembling a full transcript prepends.

Overlap with chat_poll

The same message can arrive from both, and adapters must return the same `id` for it either way. That id is the only thing a consumer has to deduplicate on – a startup backfill and the first poll after it routinely cover the same events.

Examples

```
cl <- chat_loopback()
chat_send(cl, "general", "hello")
chat_history(cl, "general")$messages
```

`chat_identity` *Construct an identity record*

Description

Construct an identity record

Usage

```
chat_identity(id, display = NA_character_, raw = NULL)
```

Arguments

`id`	The account identifier, in whatever form the transport uses. Comparable against `chat_message`'s `sender` and against the entries of its `mentions`: that comparability is the point of the field, so an adapter whose two sides disagree has a bug here rather than a choice.
`display`	Human-readable name, or NA when the adapter would have to ask the server for it. Never used for matching – it is not unique, and on most transports any account can set it to any other account's. For logs and prompts only.
`raw`	The adapter's platform-native identity payload.

Value

A list with class `chat_identity`.

Examples

```
chat_identity("@bot:example.org", display = "corteza")
```

`chat_invite` *Construct a normalized invitation record*

Description

The record `chat_poll` returns in `$invites` on adapters whose `chat_capabilities()$invites` is TRUE.

Usage

```
chat_invite(channel, inviter = NA_character_, raw = NULL)
```

Arguments

`channel`	Channel/room identifier the client has been invited to (character). Pass it to `chat_join` to accept.
`inviter`	Who issued the invitation, or `NA` when the transport does not say. NA rather than NULL, and deliberately: a consumer deciding whether to accept has to tell "someone I do not trust" from "I could not tell who", and both refuse for different reasons. NULL would collapse the second into the absence of a field.
`raw`	The adapter's platform-native payload.

Value

A list with class `chat_invite`.

No timestamp

Unlike `chat_message` and `chat_reaction`, there is no `ts`. An invitation is a standing state rather than an event at a moment, and Matrix's stripped invite state carries no reliable `origin_server_ts` to report. A field that could only ever be NA is worse than no field.

Examples

```
chat_invite("!room:example.org", inviter = "@alice:example.org")
```

chat_irc *Create an IRC chat client*

Description

Connects, registers (NICK/USER), and joins `channels`. The persistent socket buffers into `chat_poll`: each poll drains available lines, answers server PINGs, and returns PRIVMSGs as normalized messages. The cursor is a message counter.

Usage

```
chat_irc(host, port = 6667L, nick, channels = character(), realname = nick)
```

Arguments

`host`	Server hostname.
`port`	Server port (plaintext; commonly 6667).
`nick`	Nickname to register.
`channels`	Character vector of channels to join (e.g. `"#rstats"`).
`realname`	Real-name field for USER registration.

Value

A `chat_client` of class `chat_irc`.

Examples

```
## Not run:
# Requires a reachable IRC server and permission to join the channel.
cl <- chat_irc(host = "irc.example.org", nick = "example_bot",
               channels = "#example")
chat_poll(cl, timeout = 1)
chat_disconnect(cl)

## End(Not run)
```

chat_join *Join a channel*

Description

Accepts a pending invitation, or joins an open channel where the platform allows it.

Usage

```
chat_join(client, channel, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier, as carried on a chat_invite's channel or resolved by chat_resolve.
...	Adapter-specific options.

Details

The default method throws, on the same reasoning as chat_react: a join that silently does nothing leaves the caller believing it is in a room it will never hear from. Check chat_capabilities()$join.

Value

The joined channel's identifier, invisibly.

Examples

```
## Not run:
# Requires a saved Matrix configuration and access to the room.
cl <- chat_matrix(app = "mybot")
chat_join(cl, "#general:example.org")

## End(Not run)
```

---

chat_leave *Leave a channel*

Description

The inverse of chat_join: after it returns, the client stops receiving the channel's traffic, on platforms where membership is a thing at all. Capability-gated: check chat_capabilities()$leave.

Usage

```
chat_leave(client, channel, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
...	Adapter-specific options.

Value

The left channel's identifier, invisibly.

Examples

```
## Not run:
# Requires a saved Matrix configuration and a joined room.
cl <- chat_matrix(app = "mybot")
chat_leave(cl, "#general:example.org")

## End(Not run)
```

---

chat_loopback *Create a loopback chat client*

Description

Messages sent to any channel are appended to an in-memory log and come back out of `chat_poll`. The cursor is the integer position in that log.

Usage

```
chat_loopback()
```

Value

A `chat_client` of class `chat_loopback`.

Examples

```
cl <- chat_loopback()
chat_send(cl, "general", "hello")
chat_poll(cl)$messages
```

---

chat_mark_read *Mark a message as read*

Description

The default is a quiet FALSE, on `chat_typing`'s reasoning rather than `chat_react`'s: a read marker that does not appear costs a human a little context about what the bot has seen, and nothing more. Nobody is waiting on it the way they wait on an acknowledgement.

Usage

```
chat_mark_read(client, channel, message_id, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
message_id	The message to mark read, and everything before it.
...	Adapter-specific options.

Details

Write-only. Reading other participants' read state is a much larger surface – per-user, per-device, and absent entirely on some platforms – and no consumer needs it yet.

Value

TRUE if the marker was sent, FALSE otherwise, invisibly.

Examples

```
cl <- chat_loopback()
id <- chat_send(cl, "general", "hello")
chat_mark_read(cl, "general", id) # FALSE: no read markers
```

---

chat_matrix *Create a Matrix chat client*

Description

Wraps an mx.client client config (see mx.client::mx_client_load()). The sync cursor lives inside the mx.client config; with save_cursor = TRUE every poll persists it, so a restarted process resumes where it left off.

Usage

```
chat_matrix(
  app = NULL,
  path = NULL,
  save_cursor = TRUE,
  mx = NULL,
  relogin = TRUE,
  e2ee = FALSE,
  crypto_store = NULL,
  .sync = NULL,
  .extract = NULL,
  .send = NULL,
  .media = NULL,
  .typing = NULL,
  .crypto = NULL,
```

```
  .save = NULL,
  .react = NULL,
  .info = NULL,
  .members = NULL,
  .join = NULL,
  .create = NULL,
  .leave = NULL,
  .invite = NULL,
  .state = NULL,
  .get_state = NULL,
  .extract_media = NULL,
  .download = NULL,
  .channels = NULL,
  .history = NULL,
  .pending = NULL,
  .read = NULL,
  .identity = NULL,
  .edit = NULL,
  .rich = NULL,
  .avatar = NULL
)
```

Arguments

app	Application namespace passed to `mx.client::mx_client_load()`, defaulting to "chat.api" for the load. NULL (the default) is also what gets forwarded to `mx_sync_update()`, which leaves the wrapped config's own app/path attributes in charge of where the cursor is persisted. Naming an app here overrides them, so only name one when this adapter owns the config file.
path	Explicit config path, or NULL for the app default.
save_cursor	Logical. Persist the sync token after each poll, through mx.client, into the config file the wrapped client points at. Pass FALSE only if you persist `chat_poll()`'s `cursor` yourself; nothing else writes it, so a FALSE that isn't paired with a save makes every restart replay history from a frozen token.
mx	A ready mx.client client config to wrap, for consumers that already load and manage one; NULL (default) loads via app/path.
relogin	Logical. Wrap each sync in `mx.client::mx_with_relogin()`, which catches an invalidated access token, re-logs in with the stored password on the same device_id (so an E2EE identity survives), and retries once. The refreshed config lands back in the client, so the next poll and any send use the new token. FALSE lets `mx_error_M_UNKNOWN_TOKEN` propagate.
e2ee	Logical. Own Olm/Megolm state in the adapter, so `chat_send` encrypts for rooms that advertise `m.room.encryption` and `chat_poll` decrypts inbound. FALSE (the default) is the previous behaviour exactly: cleartext only, no crypto state, and `e2ee` stays FALSE in `chat_capabilities`. Requires the 'mx.crypto' package, which needs a Rust toolchain, and both `user_id` and `device_id` on the wrapped config.

Three things change when it is on. The crypto state is built on first use rather than here: from chat_poll that means key publication happens after the sync, so a relogin has already replaced a rejected token, while a chat_send that runs before any poll publishes with the token it has – the same exposure any direct send already carries, since mx_with_relogin() wraps only the sync. The sync cursor is held back until the crypto state is safe, on disk and on the client, so a sync carrying a room key is not marked consumed until that key is stored. Attachments use ciphertext uploads and encrypted file metadata with mx.client 0.3.0 and mx.crypto 0.3.0 or later. Older dependencies refuse encrypted attachments before upload; chat_capabilities reports files according to the installed helpers.

crypto_store — Character or NULL. Directory holding the pickled Olm account and sessions. NULL derives it from app via mx.client::mx_crypto_store_dir(), with a per-device subdirectory, so moving a config does not cost the device identity and two bots under one app namespace do not share an Olm account.

A Matrix device has one Olm identity for its whole life, and three things hold that. The store records the (user_id, device_id) it belongs to and refuses to open for a different one. Within a process, one device gets one context and one store, so pointing a second store at it is an error rather than a second account. And on first use the account's keys are checked against the ones the homeserver already has for that device_id – the only one of the three that survives a restart, and so the one that catches a store swapped between runs. Log in again for a new device_id rather than re-homing an existing one.

That last check distinguishes three cases: no published device is a first run, a published device whose keys match is this account, and a published device that either differs or fails signature verification is an error. A /keys/query that returns a failures map is an error too, since the empty result beside it is an unanswered question rather than an absent device. It cannot see a homeserver that omits the device deliberately, which is indistinguishable from a first run without key pinning or cross-signing.

A Windows drive-relative crypto_store ("C:store") is rejected. It names a directory relative to the current directory of that drive, which cannot be read here, so two such paths cannot be told apart and two stores could share one context. Pass an absolute path.

.sync — Testing seam: replacement for mx.client::mx_sync_update. Leave NULL in production.

.extract — Testing seam: replacement for mx.client::mx_extract_text_events. Leave NULL in production.

.send — Testing seam: replacement for mx.client::mx_send_text. Leave NULL in production.

.media — Testing seam: replacement for mx.client::mx_send_media. Leave NULL in production. Supplying all four seams together with mx lets poll and send run without mx.client installed; chat_typing() and chat_resolve() still require it.

.typing — Testing seam: replacement for mx.api::mx_typing. Leave NULL in production. It is resolved when chat_typing() is called, not here, so a NULL seam costs nothing on installs without mx.api.

.crypto	Testing seam: a named list overriding any of the adapter's four crypto operations – `init`, `encrypted`, `send`, `decrypt`. Leave NULL in production. Supplying `init` is what lets `e2ee = TRUE` be tested on a runner with neither mx.crypto nor a Rust toolchain; E2EE is the one part of this adapter with no honest way to reach a real homeserver from a test.
.save	Testing seam: replacement for `mx.client::mx_client_save`. Leave NULL in production. Only reached on an `e2ee` client, which writes the sync cursor after the crypto state rather than inside the sync.
.react	Testing seam: replacement for `mx.api::mx_react`. Leave NULL in production. Resolved when `chat_react()` is called, not here, so a NULL seam costs nothing on installs without mx.api.
.info	Testing seam: a list with `name` and `topic`, replacing `mx.api::mx_room_name` and `mx.api::mx_room_topic`. Leave NULL in production.
.members	Testing seam: replacement for `mx.api::mx_room_members`. Leave NULL in production.
.join	Testing seam: replacement for `mx.api::mx_room_join`. Leave NULL in production.
.create	Testing seam: replacement for `mx.api::mx_room_create`. Leave NULL in production.
.leave	Testing seam: replacement for `mx.api::mx_room_leave`. Leave NULL in production.
.invite	Testing seam: replacement for `mx.api::mx_room_invite`. Leave NULL in production.
.state	Testing seam: replacement for `mx.api::mx_set_state`. Leave NULL in production.
.get_state	Testing seam: replacement for `mx.api::mx_get_state`. Leave NULL in production.
.extract_media	Testing seam: replacement for `mx.client::mx_extract_media_events`. Leave NULL in production. Supplying it also flips `chat_capabilities()$attachments` on, since a seam is a media source like any other.
.download	Testing seam: replacement for `mx.api::mx_download`. Leave NULL in production.
.channels	Testing seam: replacement for `mx.api::mx_rooms`. Leave NULL in production.
.history	Testing seam: replacement for `mx.api::mx_messages`. Leave NULL in production.
.pending	Testing seam: replacement for `mx.api::mx_sync`, used by `chat_pending` for its cursorless snapshot. Leave NULL in production.
.read	Testing seam: replacement for `mx.api::mx_read_receipt`. Leave NULL in production.
.identity	Testing seam: replacement for `mx.client::mx_set_displayname`. Leave NULL in production.
.edit	Testing seam: replacement for `mx.api::mx_send` on the edit path. Leave NULL in production.

.rich	Testing seam: replacement for mx.api::mx_send on the rich-send path. Leave NULL in production.
.avatar	Testing seam: replacement for the upload-then-set pair behind chat_set_identity(avatar =) (mx.api::mx_upload and mx_set_avatar_url). Leave NULL in production.

Value

A chat_client of class chat_matrix. chat_poll on this class returns first_run and client alongside messages, cursor, and raw. first_run is TRUE when the sync started from no cursor, so the messages are a backfill baseline rather than new traffic. client is the post-sync mx.client config, which a consumer needs whenever it drives mx.api directly (read receipts, member lookups) because a relogin may have replaced the token this poll cycle.

Examples

```
## Not run:
# Requires mx.client and saved Matrix credentials for this application.
cl <- chat_matrix(app = "mybot")
chat_capabilities(cl)
chat_poll(cl, timeout = 0)

## End(Not run)
```

---

chat_matrix_config *Load a Matrix configuration*

Description

Reads the credentials an application saved, and returns them as a chat_config: a list carrying the transport's own fields plus whatever else the application stored alongside them, with the app namespace and file path attached as attributes so chat_config_save can write it back where it came from.

Usage

```
chat_matrix_config(app = NULL, path = NULL, env_var = NULL)
```

Arguments

app	Application namespace, e.g. "corteza". Decides the default path under tools::R_user_dir(app, "config").
path	Explicit file path, overriding app's default.
env_var	Name of an environment variable that, when set, overrides both.

Value

A list with class chat_config.

Why the extra fields survive

Applications keep their own settings in the same file – which accounts count as bots, who may open a private conversation, a preferred model. Those are the application's, not the transport's, and a loader that dropped them would make the file unreadable by its owner. They pass through untouched and unvalidated.

Examples

```
if (requireNamespace("mx.client", quietly = TRUE)) {
  path <- tempfile(fileext = ".json")
  cfg <- chat_config(list(server = "https://matrix.example.org",
                          token = "example-token",
                          user_id = "@bot:example.org"))
  chat_config_save(cfg, path = path)
  chat_matrix_config(path = path)
  unlink(path)
}
```

---

chat_matrix_configure *Configure a Matrix account interactively*

Description

Logs in to a homeserver, resolves the room, and writes the resulting credentials.

Usage

```
chat_matrix_configure(
  server,
  user,
  password,
  room = NULL,
  app = NULL,
  path = NULL,
  device_id = NULL,
  extra = list()
)
```

Arguments

server	Homeserver base URL.
user	Localpart or full user id.
password	Account password.
room	Room id or alias to record as the default.
app	Application namespace to save under.
path	Explicit path to save to, overriding app.

device_id	Device id to log in as. Reusing one keeps an existing E2EE identity; a new one starts a new device.
extra	Named list of application fields to store alongside the credentials.

Value

A chat_config, invisibly.

Examples

```
## Not run:
# Requires a real homeserver, account password, and access to the room.
pw <- Sys.getenv("MATRIX_PASSWORD")
cfg <- chat_matrix_configure(server = "https://matrix.example.org",
                             user = "bot", password = pw,
                             room = "#lab:example.org", app = "mybot")

## End(Not run)
```

---

chat_matrix_config_path

Where a Matrix configuration lives

Description

Where a Matrix configuration lives

Usage

```
chat_matrix_config_path(app, env_var = NULL, legacy = FALSE)
```

Arguments

app	Application namespace.
env_var	Name of an environment variable that overrides the default path, or NULL.
legacy	Return the pre-R_user_dir location instead, for an application that needs to migrate an older file.

Value

The file path (character).

Examples

```
if (requireNamespace("mx.client", quietly = TRUE)) {
  chat_matrix_config_path("demo")
}
```

chat_members *List a channel's members*

Description

Separate from chat_channel_info because it is the expensive half: a member list is unbounded where a name and a topic are two short strings, and it goes stale on a different schedule.

Usage

```
chat_members(client, channel, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
...	Adapter-specific options.

Value

Character vector of member identifiers. Empty when the channel has none; an adapter that cannot answer throws, so an empty room is never confused with an unanswerable question. Check chat_capabilities()$members first.

Examples

```
## Not run:
# Requires a saved Matrix configuration and a joined room.
cl <- chat_matrix(app = "mybot")
chat_members(cl, "#general:example.org")

## End(Not run)
```

chat_message *Construct a normalized chat message*

Description

The record every adapter's chat_poll returns.

Usage

```
chat_message(
  id,
  channel,
  sender,
  body,
  ts,
  thread = NULL,
  markup = "plain",
  kind = "message",
  self = NULL,
  mentions = NULL,
  raw = NULL,
  encrypted = FALSE,
  sender_verified = NULL,
  attachments = NULL
)
```

Arguments

id	Message identifier (character).
channel	Channel/room identifier (character).
sender	Sender identifier (character).
body	Message text (character).
ts	POSIXct timestamp of when the platform recorded the message, or NA when the transport does not report one. It is never a stand-in for the poll's wall clock: a consumer windowing or ordering by ts has to be able to tell a real event time from a missing one.
thread	Thread identifier or NULL.
markup	Source markup hint (character, e.g. "plain", "html").
kind	Message kind (character; "message" default). Contract vocabulary, not platform vocabulary: "message", "notice", "emote". Adapters translate their native type into it on the way in, the same mapping chat_send applies on the way out.
self	Logical: did this client send the message? Poll returns the bot's own traffic like any other, so a consumer that replies to inbound mail needs this to avoid answering itself. NULL when the adapter cannot tell.
mentions	Character vector of user identifiers the message explicitly mentioned (Matrix m.mentions, Slack user refs), or NULL. The signal a bot in a multi-human room gates on; body substring matching misses rich mentions that carry no plain text.
raw	The adapter's platform-native payload for this message, exactly as the transport layer handed it over. Shape is adapter-specific and may already be normalized by the transport package (Matrix hands over an extracted record, not the timeline event), so it is an escape hatch, not a guarantee of completeness.
encrypted	Logical: did this message arrive end-to-end encrypted? FALSE on transports without E2EE and on cleartext messages in rooms that have it.

`sender_verified`
: Logical: does the sender identifier bind to a device whose keys this client verified? NULL on cleartext messages, where the transport asserts the sender and there is nothing to verify. On an encrypted message FALSE is a real answer, not a missing one: the payload decrypted, but its claimed sender could not be tied to a verified device, so the identifier is the homeserver's word rather than cryptographic fact.

`attachments`
: List of `chat_attachment` records, or NULL. Inbound media: what a sent `files` = looks like from the receiving side, on adapters whose `chat_capabilities()$attachments` is TRUE.

Value

A list with class `chat_message`.

Examples

```
chat_message("m1", "general", "alice", "hello",
             ts = as.POSIXct("2026-01-01", tz = "UTC"))
```

---

chat_pending *Read standing state that is not tied to a cursor*

Description

Today: pending invitations. `chat_poll` reports an invitation when it arrives, which is no help to a client that was not running at the time – and some homeservers only report invites newer than the `since` token, so the poll loop never sees them again.

Usage

```
chat_pending(client, ...)
```

Arguments

`client`
: A `chat_client`.

`...`
: Adapter-specific options.

Details

This is a separate verb rather than a mode of `chat_poll` deliberately. Overloading the cursor would make "start from nothing" and "tell me what is standing" the same call, and a client that asked for pending invitations and thereby reset its read position would replay every channel it is in.

Value

A list with `invites`, a list of `chat_invite`.

Examples

```
## Not run:
# Requires a saved Matrix configuration and a homeserver connection.
client <- chat_matrix(app = "mybot")
pending <- chat_pending(client)
for (iv in pending$invites) chat_join(client, iv$channel)

## End(Not run)
```

`chat_poll` *Poll a chat client for new messages*

Description

Poll-shaped everywhere: long-poll transports (Matrix /sync, Telegram getUpdates) map directly; persistent-socket transports (IRC) buffer into the poll. The cursor is opaque and adapter-specific; pass the returned cursor back as `since` on the next call.

Usage

```
chat_poll(client, since = NULL, timeout = NULL, ...)
```

Arguments

`client`	A `chat_client`.
`since`	Opaque cursor from the previous poll, or NULL to start.
`timeout`	Seconds to wait for activity; NULL for the adapter default.
`...`	Adapter-specific options.

Value

A list with `messages` (list of `chat_message`) and `cursor` (opaque, for the next `since`).

Examples

```
cl <- chat_loopback()
chat_send(cl, "general", "hello")
batch <- chat_poll(cl)
batch$messages
chat_poll(cl, since = batch$cursor)$messages
```

chat_react *React to a message*

Description

Places a reaction (an emoji or short key) on an existing message.

Usage

```
chat_react(client, channel, message_id, key, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
message_id	Identifier of the message being reacted to, as returned by chat_send or carried on a chat_message's id.
key	The reaction itself. Platforms differ on what they accept: Matrix takes any string and conventionally an emoji character, Slack takes a short name without colons ("thumbsup"). Passed through unchanged, since translating between the two would have to guess.
...	Adapter-specific options.

Details

The default method throws. A reaction that silently does nothing is worse than one that fails: the caller believes it acknowledged something and no one can see that it did not. This differs from chat_typing, whose default is a quiet FALSE, because a missing typing indicator costs nothing and a missing acknowledgement can be the whole message. Check chat_capabilities()$reactions before calling on an unknown adapter.

Value

The reaction's identifier where the platform gives it one (Matrix), invisibly; TRUE where it does not (Slack).

Examples

```
## Not run:
# Requires a saved Matrix configuration and a joined room.
cl <- chat_matrix(app = "mybot")
room <- chat_resolve(cl, "#general:example.org")
id <- chat_send(cl, room, "hello")
chat_react(cl, room, id, "+1")

## End(Not run)
```

chat_reaction *Construct a normalized reaction record*

Description

The record chat_poll returns in $reactions on adapters whose chat_capabilities()$reaction_events is TRUE.

Usage

```
chat_reaction(id, channel, sender, target, key, ts, self = NULL, raw = NULL)
```

Arguments

id	The reaction's own identifier, or NULL where the platform does not give it one. Not the identifier of the message it is attached to – see target.
channel	Channel/room identifier (character).
sender	Sender identifier (character).
target	Identifier of the message being reacted to (character).
key	The reaction itself (character): an emoji on Matrix, a short name on Slack.
ts	POSIXct timestamp, or NA when the transport does not report one – the same rule chat_message follows, and for the same reason.
self	Logical: did this client place the reaction? A consumer that reacts to acknowledge needs this to avoid answering its own acknowledgement. NULL when the adapter cannot tell.
raw	The adapter's platform-native payload.

Details

Deliberately not a chat_message. A reaction has a target and no body, and the message record has a body and no target; thread is the closest slot and it means something else, so folding one into the other would make every consumer disambiguate by inspecting fields.

Value

A list with class chat_reaction.

Examples

```
chat_reaction("r1", "general", "alice", target = "m1",
              key = "+1", ts = as.POSIXct("2026-01-01", tz = "UTC"))
```

chat_relogin *Refresh this client's credentials*

Description

Forces the re-authentication that adapters otherwise perform on demand. The refreshed credentials stay inside the client.

Usage

```
chat_relogin(client, ...)
```

Arguments

client	A chat_client.
...	Adapter-specific options.

Details

The default throws rather than returning quietly. "I could not refresh" and "there was nothing to refresh" look identical to a caller that gets FALSE, and the first means the next call will fail with a stale token.

Value

TRUE, invisibly.

Examples

```
## Not run:
# Requires a saved Matrix configuration with login credentials.
cl <- chat_matrix(app = "mybot")
chat_relogin(cl)

## End(Not run)
```

chat_resolve *Resolve a human channel name to its identifier*

Description

Resolve a human channel name to its identifier

Usage

```
chat_resolve(client, name, ...)
```

Arguments

client	A chat_client.
name	Channel name, alias, or identifier.
...	Adapter-specific options.

Value

The adapter-native channel identifier (character).

Examples

```
chat_resolve(chat_loopback(), "general")
```

---

chat_send *Send a message through a chat client*

Description

Send a message through a chat client

Usage

```
chat_send(
  client,
  channel,
  text,
  markup = c("plain", "markdown"),
  thread = NULL,
  reply_to = NULL,
  identity = NULL,
  files = NULL,
  kind = "message",
  notify = TRUE,
  rich = NULL,
  ...
)
```

Arguments

client	A chat_client.
channel	Channel/room identifier (adapter-native or resolved via chat_resolve).
text	Message text.
markup	"plain" or "markdown"; adapters render markdown to their dialect (HTML for Matrix, mrkdwn for Slack, stripped for IRC).
thread	Thread identifier to post into, or NULL.

reply_to	Message id being replied to, or NULL.
identity	Optional per-message identity override (list(name =, icon =)) where the platform supports it.
files	Character vector of file paths to attach, or NULL.
kind	Message kind; "message" (default) or an adapter-understood alternative (e.g. "notice", "emote").
notify	Logical; FALSE requests a silent delivery where supported.
rich	Adapter-native markup for the platforms that accept it, or NULL. Matrix takes an HTML fragment and sends it as formatted_body, Telegram sends it with the HTML parse mode; adapters whose chat_capabilities()$rich_markup is empty ignore it. text is still required and still has to stand on its own. It is what a client that cannot render the markup shows, what a push notification carries, and what every other transport gets – so a rich that holds the real content and a text that says "see above" is a message half the room cannot read. Ignored rather than refused where unsupported, on chat_typing's reasoning: the text is the message and the markup is decoration, so losing it costs presentation and nothing else.
...	Adapter-specific options.

Value

Character vector of the message ids this call created, in the order they were sent, invisibly. Usually length one. A platform that splits a send into several events returns one id per event: the Matrix adapter sends each attachment as its own event, so a send with files returns the attachment ids followed by the text id. Callers that track their own traffic by id must handle every element, or an unclaimed event reads as somebody else's message.

Examples

```
cl <- chat_loopback()
id <- chat_send(cl, "general", "hello", markup = "plain")
chat_send(cl, "general", "a reply", thread = id)
chat_poll(cl)$messages
```

---

chat_set_identity *Set this client's persistent identity*

Description

The account's own display name and, where the platform lets a bot set one, its picture, as everyone in every channel sees them until they are changed again. Distinct from chat_send's identity argument, which decorates a single message on platforms that allow it.

Usage

```
chat_set_identity(client, display = NULL, avatar = NULL, ...)
```

Arguments

client	A chat_client.
display	New display name, or NULL to leave it as it is.
avatar	Path to an image file for the account's picture, or NULL to leave it. Set where the platform lets a bot change its own picture: Matrix uploads the file and points the profile at it. An adapter whose platform sets a bot's picture outside its API ignores it, on chat_typing's reasoning, and answers FALSE when it was all that was asked for.
...	Adapter-specific options.

Details

Owning this matters beyond tidiness. On Matrix the rename is an authenticated call that can rotate the access token underneath the caller, and a consumer that made that call itself had to notice the rotation and get the new token back into its client – usually via whatever file both of them happened to share. Behind the contract the rotation lands in the client that performed it, and nothing outside has to know it happened.

Value

TRUE if the identity was changed, invisibly; FALSE when nothing was asked for that the adapter can set.

Examples

```
## Not run:
# Requires a saved Matrix configuration and account credentials.
cl <- chat_matrix(app = "mybot")
chat_set_identity(cl, "Example Bot")
chat_set_identity(cl, avatar = "face.png")

## End(Not run)
```

---

chat_set_state *Set durable typed state on a channel*

Description

Attaches a typed, durable piece of metadata to a channel, readable by every client in it and replaced by the next write to the same type and state_key. On Matrix this is a state event; most platforms have no equivalent, which is why it is capability-gated: check chat_capabilities()$set_state.

Usage

```
chat_set_state(client, channel, type, content, state_key = "", ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
type	Character. Namespaced event type, e.g. "m.room.topic" or a reversed-domain custom type.
content	Named list. The state content. A write replaces the whole content for its type/state_key pair; there is no merge.
state_key	Character. Sub-key within the type. Most state is keyed by the empty string, the default.
...	Adapter-specific options.

Details

The default method throws, on chat_react's reasoning: a state write that silently did nothing leaves the caller believing a marker is set that no reader will ever see.

Value

The state event's identifier where the platform gives one (Matrix), invisibly; TRUE where it does not.

Examples

```
cl <- chat_loopback()
chat_set_state(cl, "general", "m.room.topic", list(topic = "Planning"))
chat_get_state(cl, "general", "m.room.topic")
```

---

chat_slack *Create a Slack chat client*

Description

Requires the suggested **slackr** package and a bot token. Channel names are normalized to bare names (no #), matching what slackr's channel translation accepts.

Usage

```
chat_slack(
  channels = character(),
  token = Sys.getenv("SLACK_TOKEN"),
  user_token = Sys.getenv("SLACK_USER_TOKEN"),
  username = NULL,
  .history = NULL,
```

```
  .post = NULL,
  .react = NULL,
  .api = NULL
)
```

Arguments

channels	Character vector of channels to poll.
token	Bot token; defaults to the SLACK_TOKEN environment variable.
user_token	User token (xoxp-...) for posting and resolving identity as a real workspace member rather than the bot; defaults to the SLACK_USER_TOKEN environment variable. Optional – leave unset (empty string) if as_user is never used.
username	Default display-name override for sends, or NULL (default) to post as the bot's own identity.
.history	Testing seam: replacement for slackr::slackr_history. Leave NULL in production.
.post	Testing seam: replacement for slackr::slackr_msg. Leave NULL in production; when both seams are supplied the slackr package is not required.
.react	Testing seam: replacement for slackr::call_slack_api. Leave NULL in production. Resolved when chat_react() is called, not here.
.api	Testing seam: replacement for slackr::call_slack_api on the read paths (chat_channel_info(), chat_members()). Leave NULL in production.

Details

Sends explicitly suppress slackr's SLACK_USERNAME / SLACK_ICON_EMOJI environment defaults: a plain chat_send() posts as the bot's own identity, and authorship is only overridden through username here or chat_send(identity =) (both need the chat:write.customize scope).

user_token is a different kind of authorship than identity: identity/username relabel a bot's own post with a cosmetic name and icon, while a user token (xoxp-..., obtained via a Slack app's User Token Scopes rather than its Bot Token Scopes) authenticates as an actual workspace member, so chat_send(..., as_user = TRUE) and chat_whoami(..., as_user = TRUE) post and resolve identity as that member – Slack shows their real name and photo, not a bot profile. Optional: leave unset if you only ever post as the bot.

A post made as_user = TRUE is the member's own message in every respect, including to this client's own chat_poll: Slack messages carry no self, so nothing distinguishes it from something the member typed, and a consumer that replies to the member's traffic will reply to it.

Value

A chat_client of class chat_slack.

Examples

```
## Not run:
# Requires slackr, SLACK_TOKEN, and a channel the token can access.
cl <- chat_slack(channels = "C0123456789")
```

```
chat_send(cl, "C0123456789", "hello")

## End(Not run)
```

`chat_telegram` *Create a Telegram chat client*

Description

Requires the suggested **httr** package and a bot token from BotFather, or a `telegram::TGBot` that carries one.

Usage

```
chat_telegram(
  token = Sys.getenv("TELEGRAM_BOT_TOKEN"),
  timeout = 30L,
  api_url = "https://api.telegram.org",
  bot = NULL,
  .api = NULL,
  .download = NULL
)
```

Arguments

`token`	Bot token; defaults to the `TELEGRAM_BOT_TOKEN` environment variable.
`timeout`	Long-poll wait in seconds, used by `chat_poll` when it is given no `timeout`.
`api_url`	Base URL of the Bot API. The default is Telegram's; a local Bot API server takes its own. Ignored when `bot` is given, since the class fixes the host.
`bot`	A `telegram::TGBot` from the suggested **telegram** package, or NULL. When given, every request goes through its public `req()` method, so its proxy settings apply and `token` may be left empty: the object holds its own. The class's verbs are not used, only its transport. Its `getUpdates()` can neither long-poll nor choose update kinds, its parser flattens updates into data frames, and it has no edit, reaction, chat or leave methods; `req()` is what carries this adapter. Attachments are fetched from the URL its `getFile()` returns.
`.api`	Testing seam: replacement for the HTTP layer, a `function(method, params, files)` returning the parsed response (`list(ok =, result =)`). `params` arrives already in wire form: NULLs dropped, logicals as `"true"`/`"false"`, numbers as plain digits. Leave NULL in production.
`.download`	Testing seam: replacement for the file fetch, a `function(file_id, file_path, dest)` writing the bytes behind a getFile answer to `dest`. Leave NULL in production; when both seams are supplied neither httr nor a bot is required.

Details

Channels are chat identifiers as Telegram reports them – a positive number for a private chat, a negative one for a group or channel – always as character. chat_resolve turns a public @@username into one.

The first poll returns whatever updates Telegram is still holding for the bot (it keeps them for 24 hours). That is the mail that arrived while the bot was down rather than channel history, so it comes out as ordinary traffic. Passing the returned cursor back as since confirms it; Telegram re-sends anything unconfirmed.

Value

A chat_client of class chat_telegram.

Examples

```
## Not run:
# Requires httr, TELEGRAM_BOT_TOKEN, and access to the target chat.
cl <- chat_telegram()
chat_whoami(cl)
chat_send(cl, "@example_channel", "hello")

## End(Not run)
```

---

chat_typing *Signal typing state in a channel*

Description

Capability-gated: the default method is a no-op so adapters without typing indicators need not implement it.

Usage

```
chat_typing(client, channel, on = TRUE, ...)

## S3 method for class 'chat_matrix'
chat_typing(client, channel, on = TRUE, timeout = 30, ...)
```

Arguments

client	A chat_client.
channel	Channel/room identifier.
on	Logical.
...	Adapter-specific options.
timeout	Seconds the typing indicator should stand before the homeserver clears it, for on = TRUE. Seconds, not milliseconds: chat_poll() takes seconds too, and the adapter converts at the mx.api boundary. A caller running a slow model turn wants a long one (say 120) and an explicit on = FALSE when the turn ends.

Value

TRUE if the signal was sent, FALSE otherwise, invisibly.

Examples

```
cl <- chat_loopback()
chat_typing(cl, "general") # FALSE: loopback has no typing indicator
```

---

chat_whoami *Who is this client logged in as?*

Description

The client's own account, as the transport identifies it.

Usage

```
chat_whoami(client, ...)
```

Arguments

client	A chat_client.
...	Adapter-specific options.

Details

The default method throws rather than guessing. Every use of an identity is a comparison – is this message mine, did someone address me – and a wrong answer to either is silent: the bot answers itself in a loop, or never answers anyone. An adapter that cannot say who it is should say so.

Value

A chat_identity.

Examples

```
chat_whoami(chat_loopback())
```

Index

∗ **package**
chat.api-package, 3

chat.api *(*chat.api-package*)*, 3
chat.api-package, 3
chat_addressed, 3, *5*
chat_attachment, 4, *5*, *11*, *29*
chat_capabilities, *3*, 5, *21, 22*
chat_channel_create, *5*, 6, *8*
chat_channel_info, *5*, 7, *27*
chat_channel_invite, *5*, 8
chat_channels, 6
chat_config, *9*, 9, *26*
chat_config_save, 9, *24*
chat_disconnect, 10
chat_download, 11
chat_edit, 12
chat_get_state, *5*, 13
chat_history, 14
chat_identity, 15, *41*
chat_invite, 16, *18*, *29*
chat_irc, *3*, 17
chat_join, *5*, *8*, *16*, 17, *18*
chat_leave, *5*, 18
chat_loopback, *3*, 19
chat_mark_read, 19
chat_matrix, *3*, 20
chat_matrix_config, 24
chat_matrix_config_path, 26
chat_matrix_configure, 25
chat_members, *5*, *7*, 27
chat_message, *3, 4*, *11*, *14–16*, 27, *31, 32*
chat_pending, *23*, 29
chat_poll, *5, 6*, *8*, *14*, *16, 17*, *19*, *21, 22*, *24*, *27*, *29*, 30, *32*, *38, 39*
chat_react, *5*, *11*, *18, 19*, 31, *37*
chat_reaction, *16*, 32
chat_relogin, 33
chat_resolve, *18*, 33, *34*, *40*
chat_send, *12*, *21, 22*, *28*, *31*, 34, *35*
chat_set_identity, *5*, 35
chat_set_state, *5*, *13*, 36
chat_slack, *3*, 37
chat_telegram, *3*, 39
chat_typing, *19*, *31*, *35, 36*, 40
chat_whoami, *5*, 41