

Package ‘shinychat’

September 9, 2026

Title Chat UI Component for 'shiny'

Version 0.5.0

Description Provides a scrolling chat interface with multiline input, suitable for creating chatbot apps based on Large Language Models (LLMs). Designed to work particularly well with the 'ellmer' R package for calling LLMs.

License MIT + file LICENSE

URL <https://posit-dev.github.io/shinychat/r/>,
<https://github.com/posit-dev/shinychat>

BugReports <https://github.com/posit-dev/shinychat/issues>

Imports base64enc, bslib (>= 0.12.0), cli, coro, ellmer (>= 0.4.1), fastmap, htmltools, jsonlite, lifecycle, promises (>= 1.3.2), R6 (>= 2.5.0), rlang (>= 1.2.0), S7, shiny (>= 1.10.0)

Suggests chromote, covr, knitr, later, rmarkdown, shinytest2, testthat (>= 3.0.0), withr

VignetteBuilder knitr, rmarkdown

Config/Needs/website tidyverse/tidytemplate, rmarkdown, weathR, gt, glue, bsicons, reactable, htmlwidgets

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation no

Author Joe Cheng [aut],
Carson Sievert [aut],
Garrick Aden-Buie [aut, cre] (ORCID:
<<https://orcid.org/0000-0002-7111-0077>>),
Barret Schloerke [aut] (ORCID: <<https://orcid.org/0000-0001-9986-114X>>),
Posit Software, PBC [cph, fnd] (ROR: <<https://ror.org/03wc8by49>>)

Maintainer Garrick Aden-Buie <garrick@adenbuie.com>

Repository CRAN

Date/Publication 2026-09-09 18:50:02 UTC

Contents

chat_app	2
chat_append	7
chat_append_message	11
chat_attachment	13
chat_clear	14
chat_drawer	15
chat_drawer_hide	16
chat_drawer_show	17
chat_drawer_toggle	17
chat_drawer_update	18
chat_enable_history	19
chat_get_greeting	20
chat_greeting	20
chat_nav_panel	22
chat_restore	23
chat_set_greeting	25
chat_sidebar	27
chat_ui	28
chat_ui_history	35
ContentSlashCommand	36
contents_shinychat	37
ConversationStore	39
FileConversationStore	41
history_options	43
markdown_stream	44
output_markdown_stream	45
page_chat	47
page_chat_theme	51
tool_result_display	51
update_chat_user_input	53
Index	55

chat_app	<i>Open a live chat application in the browser</i>
----------	--

Description

Create a simple Shiny app for live chatting using an [ellmer::Chat](#) object. Note that these functions will mutate the input `client` object as you chat because your turns will be appended to the history.

The app created by `chat_app()` is suitable for interactive use by a single user. For multi-user Shiny apps, use [chat_ui\(\)](#) and `chat_server()` and be sure to create a new chat client for each user session.

Usage

```
chat_app(
  client,
  ...,
  title = NULL,
  icon = NULL,
  window_title = NULL,
  id = "chat",
  greeting = NULL,
  history = TRUE,
  bookmark_store = "url",
  app_options = list()
)

chat_server(
  id,
  client,
  greeting = NULL,
  history = TRUE,
  bookmark_on_input = lifecycle::deprecated(),
  bookmark_on_response = lifecycle::deprecated(),
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

client	A chat object created by ellmer , e.g. <code>ellmer::chat_openai()</code> and friends.
...	Named arguments passed to <code>page_chat()</code> .
title	The title displayed in the page header. If NULL (the default), a "{model} ({provider})" title is derived from client.
icon	Optional UI displayed before title. See <code>page_chat()</code> .
window_title	The browser-window title. If NULL (the default), uses "shinychat {model} {date}" derived from client.
id	The ID shared by <code>page_chat()</code> and <code>chat_server()</code> .
greeting	Optional greeting to set when the module initializes. Accepts a static value (string, <code>htmltools::HTML()</code> , <code>htmltools::tagList()</code> , or <code>chat_greeting()</code>) or a function that generates the greeting dynamically. See the Greeting section below for details.
history	Conversation history configuration. TRUE (default) enables history with default settings; FALSE disables it; pass a <code>history_options()</code> object to customise storage, identity, titling, or hooks.
bookmark_store	The bookmarking store to use for the app. Passed to <code>enableBookmarking</code> in <code>shiny::shinyApp()</code> . Defaults to "url", which uses the URL to store the chat state. URL-based bookmarking is limited in size; use "server" to store the state on the server side without size limitations; or disable bookmarking by setting this to "disable".

app_options	A list passed to the options argument of <code>shiny::shinyApp()</code> .
bookmark_on_input	A logical value determines if the bookmark should be updated when the user submits a message. Default is TRUE.
bookmark_on_response	A logical value determines if the bookmark should be updated when the response stream completes. Default is TRUE.
session	The Shiny session. Defaults to the current reactive domain.

Value

- `chat_app()` returns a `shiny::shinyApp()` object.
- `chat_server()` includes the shinychat server logic, and returns an environment containing:
 - `last_input`: A reactive value containing the last user input (a string when attachments are disabled, a list of ellmer Content objects when enabled).
 - `last_turn`: A reactive value containing the last assistant turn.
 - `update_user_input()`: A function to update the chat input or submit a new user input. Takes the same arguments as `update_chat_user_input()`, except for `id` and `session`, which are supplied automatically.
 - `append()`: A function to append a new message to the chat UI. Takes the same arguments as `chat_append()`, except for `id` and `session`, which are supplied automatically.
 - `clear()`: A function to clear the chat client turns and the chat UI. It optionally takes a list of messages used to initialize the chat after clearing. `messages` should be a list of messages, where each message is a list with `role` and `content` fields. The `client_history` argument controls how the chat client's history is updated after clearing. It can be one of: "clear" the chat history; "set" the chat history to messages; "append" messages to the existing chat history; or "keep" the existing chat history. `clear()` is unavailable when conversation history is enabled; use `new_chat()` instead.
 - `new_chat()`: A function to save the current conversation and start a new one by clearing the chat client's turns and chat UI, resetting the active conversation, and updating the history drawer. It is available only when conversation history is enabled. `new_chat(greeting = TRUE)` also clears the greeting and requests a new one. `new_chat()` errors while a response is streaming; wait for it to complete or stop it first.
 - `set_greeting()`: A function to set, stream, or clear the chat greeting. Pass a `chat_greeting()` object, a plain string, or NULL to clear. Streaming greetings run inside an `shiny::ExtendedTask` so the session stays responsive; if called while a greeting is already streaming, the new greeting is queued. If the greeting has already been dismissed, calling `set_greeting()` updates the content but does not make it visible again; call `clear(greeting = TRUE)` first to show a new greeting after dismissal.
 - `status`: A reactive value indicating the current chat interaction state. Returns "idle" when no response is in progress, or "streaming" while a response is actively being received.
 - `history`: A namespace for managing conversation-history callbacks and persistence. `saved <- chat_module$history$save()` saves only the existing active conversation and returns whether it was saved. Storage and bookmark errors propagate to the caller.

- `last_error`: A reactive value holding the condition from the most recent response if it failed, and `NULL` otherwise. Both a finished and a failed response read as "idle" in status, so this is what tells them apart. Responses only: a greeting streams from its own task, and an error raised by a slash command handler is reported as a notification, so neither appears here.
- `client`: The current chat client object (an active binding that always reflects the latest client, even after `set_client()` is called).
- `history$conversation_id()`: A reactive expression returning the active conversation ID: `NULL` when history is disabled or the chat is still an empty draft, otherwise the ID allocated on the first user submission – before the model call – that the saved conversation record carries. The ID is stable across retries, restores, conversation switches, and `set_client()` calls. The ID is also handed to the client (via its `conversation_id` binding, when supported), which records it as the `gen_ai.conversation.id` attribute on its own OpenTelemetry spans.
- `set_client(new_client, sync = TRUE)`: Replace the chat client used by the module. When `sync` is `TRUE` (the default), the new client inherits conversation turns, system prompt, and tools from the previous client so the conversation continues seamlessly. Set `sync = FALSE` to use the new client as-is. If a response is currently streaming, the swap is deferred until the stream completes. If called multiple times while streaming, only the most recent new client is used.
- `slash_command(name, description, handler, ..., echo, force)`: Register a slash command. `handler` is required: pass a function (taking 0 or 1 argument), or `NULL` for a client-side command handled in JavaScript via the `shiny:chat-slash-command` DOM event. A handler that takes one argument receives a [ContentSlashCommand](#) object (not a plain string). See [ContentSlashCommand](#) for details on how to use this object to preserve the original command text across bookmarks. `echo` controls whether invoking the command is echoed as a user message and awaits a response; it defaults to `TRUE` when a handler is given and `FALSE` otherwise (set `echo = FALSE` for a handler that only performs side effects). Returns a function that removes the command. Errors if a command with the same name is already registered unless `force = TRUE`.

Functions

- `chat_app()`: A simple Shiny app for live chatting. Note that this app is suitable for interactive use by a single user; do not use `chat_app()` in a multi-user Shiny app context.
- `chat_server()`: Wire up batteries-included chat server logic in a Shiny session. Pair with [chat_ui\(\)](#) by passing it the same id; see *Pairing with chat_server()* in [chat_ui\(\)](#) for the top-level and module-based patterns.

Migration

... now configures [page_chat\(\)](#) instead of [shiny::shinyApp\(\)](#). Pass Shiny app options through `app_options`, and use `bookmark_store` instead of `enableBookmarking`. To customize `onStart` or `uiPattern`, compose [page_chat\(\)](#) and [chat_server\(\)](#) manually.

This is a breaking change: ... no longer accepts arguments for [shiny::shinyApp\(\)](#), including `options`, `enableBookmarking`, `onStart`, and `uiPattern`.

Greeting

When greeting is a **function**, it is called each time the `greeting_requested` event fires — on first view when the chat is empty, and again after `clear(greeting = TRUE)`. The function should return a `chat_greeting()` (typically wrapping a stream). Static values (strings, `chat_greeting()` objects) are set once at init and do not regenerate.

The function signature determines what is passed. Currently the only recognized argument is `client`.

`function(client)` (recommended). A clone of the `client` with its turn history wiped is passed as `client`. This avoids manually creating and configuring a separate client:

```
chat_server("chat", client, greeting = function(client) {
  stream <- client$stream_async("Generate a short welcome message.")
  chat_greeting(stream)
})
```

`function()` (zero arguments). You create and manage your own client:

```
chat_server("chat", client, greeting = function() {
  greeter <- ellmer::chat_openai(model = "gpt-4o")
  stream <- greeter$stream_async("Generate a short welcome message.")
  chat_greeting(stream)
})
```

Static value. Set once; does not regenerate after `clear()`:

```
chat_server("chat", client, greeting = "## Welcome!\n\nHow can I help?")
```

The returned `set_greeting()` helper is available for cases where you need to set a greeting outside the greeting lifecycle.

Examples

```
## Not run:
# Interactive in the console ----
client <- ellmer::chat_anthropic()
chat_app(client)

# Inside a Shiny app ----
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  titlePanel("shinychat example"),

  layout_columns(
    card(
      card_header("Chat with Claude"),
      chat_ui(
```

```

      "claude",
      greeting = "Hi! Use this chat interface to chat with Anthropic's `claude-3-5-sonnet`."
    ),
  ),
  card(
    card_header("Chat with ChatGPT"),
    chat_ui(
      "openai",
      greeting = "Hi! Use this chat interface to chat with OpenAI's `gpt-4o`."
    )
  )
)
)
)

server <- function(input, output, session) {
  claude <- ellmer::chat_anthropic(model = "claude-3-5-sonnet-latest") # Requires ANTHROPIC_API_KEY
  openai <- ellmer::chat_openai(model = "gpt-4o") # Requires OPENAI_API_KEY

  chat_server("claude", claude)
  chat_server("openai", openai)
}

shinyApp(ui, server)

## End(Not run)

```

chat_append

*Append an assistant response (or user message) to a chat control***Description**

The `chat_append` function appends a message to an existing `chat_ui()`. The response can be a string, string generator, string promise, or string promise generator (as returned by the 'ellmer' package's `chat`, `stream`, `chat_async`, and `stream_async` methods, respectively).

This function should be called from a Shiny app's server. It is generally used to append the client's response to the chat, while user messages are added to the chat UI automatically by the front-end. You'd only need to use `chat_append(role="user")` if you are programmatically generating queries from the server and sending them on behalf of the user, and want them to be reflected in the UI.

Usage

```

chat_append(
  id,
  response,
  role = c("assistant", "user"),
  icon = NULL,
  session = getDefaultReactiveDomain()
)

```

Arguments

<code>id</code>	The ID of the chat element
<code>response</code>	The message or message stream to append to the chat element. The actual message content can one of the following: • A string, which is interpreted as markdown and rendered to HTML on the client. – To prevent interpreting as markdown, mark the string as <code>htmltools::HTML()</code>. • A UI element. – This includes <code>htmltools::tagList()</code>, which takes UI elements (including strings) as children. Strings inside a <code>tagList</code> are literal text (HTML-escaped), not markdown. Use <code>htmltools::HTML()</code> for trusted raw HTML strings.
<code>role</code>	The role of the message (either "assistant" or "user"). Defaults to "assistant".
<code>icon</code>	An optional icon to display next to the message, currently only used for assistant messages. The icon can be any HTML element (e.g., an <code>htmltools::img()</code> tag) or a string of HTML. Pass <code>FALSE</code> to remove the icon for this message, or <code>TRUE</code> to use the default icon.
<code>session</code>	The Shiny session object

Value

Returns a promise that resolves to the contents of the stream, or an error. This promise resolves when the message has been successfully sent to the client; note that it does not guarantee that the message was actually received or rendered by the client. The promise rejects if an error occurs while processing the response (see the "Error handling" section).

Error handling

If the response argument is a generator, promise, or promise generator, and an error occurs while producing the message (e.g., an iteration in `stream_async` fails), the promise returned by `chat_append` will reject with the error. If the `chat_append` call is the last expression in a Shiny observer, `shiny-chat` will log the error message and show a message that the error occurred in the chat UI.

Asides

An aside is a small pill that appears at the end of the paragraph or list item it's attached to, showing a popover on hover, click, or keyboard focus. Create one by writing (or prompting an LLM to write) an inline `<shiny-aside>` tag anywhere in a block's markdown; the tag's content becomes the popover body:

- `<shiny-aside label="a source name" url="https://...">markdown shown in the popover</shiny-aside>`

`label` controls the text on the identity chip. A safe url makes the source heading in the popover a link. It also supplies a derived favicon unless `icon` overrides it. Without a `label`, the aside falls back to a plain numbered marker. The body is ordinary markdown: inline for a one-liner, or — by separating it with blank lines — a rich block body (paragraphs, lists, code) shown in the popover. Labeled asides in the same paragraph or list item collapse into one pill, with each aside kept as a

separate popover page. Each unlabeled aside remains a separate numbered pill. The grouped pill shows a +N overflow count only when its labeled asides have different labels. Asides that share one label use a single face with no count.

Set these CSS properties on the chat container to style aside markers:

- `--shiny-chat-aside-marker-color`
- `--shiny-chat-aside-marker-hover-color`
- `--shiny-chat-aside-marker-bg`
- `--shiny-chat-aside-marker-hover-bg`
- `--shiny-chat-aside-marker-font-family`

`grounded-span` identifies the answer text that is related to an aside. Its value must exactly match text before the tag in the same paragraph or list item. When the popover opens, shinychat highlights the most recent match. If the value does not match, no text is highlighted.

Long content wraps and scrolls within the viewport. The popover keeps the nearest scoped Bootstrap theme. In a paged popover, page changes are announced to assistive technology without repeating the body.

Set `display="compact"` to show a compact numbered reference in the message. The popover retains the source label. Compact asides in the same paragraph or list item share a marker, such as [2, 3]. To style only compact markers, set the CSS properties above on `[data-shinychat-aside-display="compact"]`.

The favicon is fetched at render time from a third-party service (DuckDuckGo's icon service), which receives the cited site's hostname. To avoid that request — for privacy, or for offline/air-gapped deployments — set the `SHINYCHAT_ASIDE_FAVICON` environment variable to `false`. You can still set `icon` to a URL you control; an explicit icon bypasses the lookup entirely.

A labeled aside with a grounded span and a one-line body:

```
chat_append(
  "chat",
  paste0(
    "Hub motors are cheaper",
    paste0(
      '<shiny-aside label="eBicycles" ',
      'url="https://ebicycles.example/hub-vs-mid-drive" ',
      'grounded-span="Hub motors are cheaper">'
    ),
    "[Hub Motor vs. Mid-Drive Motor Differences Explained]",
    "(https://ebicycles.example/hub-vs-mid-drive)",
    "</shiny-aside>",
    ", and ideal for flatter terrain."
  )
)
```

Compact labeled asides that share one numbered marker:

```
chat_append(
  "chat",
```

```

paste0(
  "Revenue is recognized at shipment",
  '<shiny-aside display="compact" label="Revenue policy">',
  "Exact revenue policy.</shiny-aside>",
  " and records are retained for 30 days",
  '<shiny-aside display="compact" label="Retention policy">',
  "Exact retention policy.</shiny-aside>."
)
)

```

Two asides cited in the same sentence collapse into one pill (the first source's label becomes the face, with a "+1" overflow):

```

chat_append(
  "chat",
  paste0(
    "Hub motors are cheaper",
    '<shiny-aside label="eBicycles" url="https://ebicycles.example">...</shiny-aside>',
    '<shiny-aside label="WIRED" url="https://wired.example">...</shiny-aside>',
    ", and ideal for flatter terrain."
  )
)

```

A label-less aside with a rich block body (falls back to a plain numbered pill):

```

chat_append(
  "chat",
  paste0(
    "Battery quality matters more than raw power",
    "<shiny-aside>\n\n",
    "**Methodology**\n\n",
    "- 40 commuter e-bike models\n",
    "- released in 2024\n\n",
    "</shiny-aside>"
  )
)

```

Examples

```

library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Dumbest chatbot in the world: ignores user input and chooses
# a random, vague response.
fake_chatbot <- async_generator(function(input) {
  responses <- c(
    "What does that suggest to you?",

```

```

    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  await(async_sleep(1))
  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(
  chat_ui("chat", fill = TRUE)
)

server <- function(input, output, session) {
  observeEvent(input$chat_user_input, {
    response <- fake_chatbot(input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)

```

chat_append_message *Low-level function to append a message to a chat control*

Description

For advanced users who want to control the message chunking behavior. Most users should use [chat_append\(\)](#) instead.

Usage

```

chat_append_message(
  id,
  msg,
  chunk = TRUE,
  operation = c("append", "replace"),
  icon = NULL,
  session = getDefaultReactiveDomain()
)

```

Arguments

<code>id</code>	The ID of the chat element
<code>msg</code>	The message to append. Should be a named list with <code>role</code> and <code>content</code> fields. The <code>role</code> field should be either "user" or "assistant". The <code>content</code> field should be a string containing the message content, in Markdown format.
<code>chunk</code>	Whether <code>msg</code> is just a chunk of a message, and if so, what type. If <code>FALSE</code> , then <code>msg</code> is a complete message. If "start", then <code>msg</code> is the first chunk of a multi-chunk message. If "end", then <code>msg</code> is the last chunk of a multi-chunk message. If <code>TRUE</code> , then <code>msg</code> is an intermediate chunk of a multi-chunk message. Default is <code>FALSE</code> .
<code>operation</code>	The operation to perform on the message. If "append", then the new content is appended to the existing message content. If "replace", then the existing message content is replaced by the new content. Ignored if <code>chunk</code> is <code>FALSE</code> .
<code>icon</code>	An optional icon to display next to the message, currently only used for assistant messages. The icon can be any HTML element (e.g., <code>htmltools::img()</code> tag) or a string of HTML. Pass <code>FALSE</code> to remove the icon for this message, or <code>TRUE</code> to use the default icon.
<code>session</code>	The Shiny session object

Value

Returns nothing (`invisible(NULL)`).

Examples

```
library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Dumbest chatbot in the world: ignores user input and chooses
# a random, vague response.
fake_chatbot <- async_generator(function(id, input) {
  responses <- c(
    "What does that suggest to you?",
    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  # Use low-level chat_append_message() to temporarily set a progress message
  chat_append_message(id, list(role = "assistant", content = "_Thinking..._"))
  await(async_sleep(1))
  # Clear the progress message
  chat_append_message(id, list(role = "assistant", content = ""), operation = "replace")

  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
```

```

      yield(chunk)
      await(async_sleep(0.02))
    }
  })

  ui <- page_fillable(
    chat_ui("chat", fill = TRUE)
  )

  server <- function(input, output, session) {
    observeEvent(input$chat_user_input, {
      response <- fake_chatbot("chat", input$chat_user_input)
      chat_append("chat", response)
    })
  }

  shinyApp(ui, server)

```

chat_attachment	<i>Create an attachment from a local file path</i>
-----------------	--

Description

Reads a file, base64-encodes its contents, and returns a list in the format expected by the `attachments` argument of `update_chat_user_input()`.

Usage

```
chat_attachment(path, mime = NULL, name = NULL)
```

Arguments

<code>path</code>	Path to the file. The file must exist.
<code>mime</code>	MIME type of the file. When <code>NULL</code> (default), guessed from the file extension. Raises an error if the extension is unrecognised.
<code>name</code>	Filename shown in the attachment chip. Defaults to <code>basename(path)</code> .

Value

A list with elements `mime`, `name`, `size`, and `data_url`, ready to pass as an element of the `attachments` argument of `update_chat_user_input()`.

chat_clear	<i>Clear all messages from a chat control</i>
------------	---

Description

Removes all messages from the chat UI. Set `greeting = TRUE` to also clear the greeting, which re-triggers `greeting_requested` (see the **Greeting** section in [chat_ui\(\)](#)).

This is a UI-level primitive: it does not clear an ellmer client's turns or reset conversation history managed by [chat_server\(\)](#). For a full reset of a managed chat, call the `new_chat()` method on the value returned by [chat_server\(\)](#).

Usage

```
chat_clear(id, greeting = FALSE, session = getDefaultReactiveDomain())
```

Arguments

<code>id</code>	The ID of the chat element
<code>greeting</code>	If TRUE, also clears the greeting. When the greeting is cleared, <code>greeting_requested</code> will fire again (if the chat is visible), allowing the server to generate a new greeting.
<code>session</code>	The Shiny session object

Examples

```
library(shiny)
library(bslib)

ui <- page_fillable(
  chat_ui("chat", fill = TRUE),
  actionButton("clear", "Clear chat")
)

server <- function(input, output, session) {
  observeEvent(input$clear, {
    chat_clear("chat")
  })

  observeEvent(input$chat_user_input, {
    response <- paste0("You said: ", input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

```

library(shiny)
library(bslib)
library(shinychat)

# Regenerate greeting on clear
ui <- page_fillable(
  chat_ui("chat"),
  actionButton("new_chat", "New chat")
)

server <- function(input, output, session) {
  observeEvent(input$chat_greeting_requested, {
    chat_set_greeting("chat", "## Welcome!\n\nHow can I help?")
  })

  observeEvent(input$new_chat, {
    # Clearing with greeting = TRUE triggers greeting_requested again
    chat_clear("chat", greeting = TRUE)
  })

  observeEvent(input$chat_user_input, {
    chat_append("chat", paste0("You said: ", input$chat_user_input))
  })
}

shinyApp(ui, server)

```

chat_drawer

Create a chat drawer configuration

Description

An drawer displays UI content adjacent to a chat interface, such as a preview, a generated report, or a detail view. Use `chat_drawer()` to supply its initial content and layout to the drawer argument of `chat_ui()` or `page_chat()`. Update the panel later with the other drawer functions.

Usage

```
chat_drawer(..., title = NULL, width = 400, open = TRUE, resizable = TRUE)
```

Arguments

<code>...</code>	UI content to display in the drawer.
<code>title</code>	An optional drawer title.
<code>width</code>	The initial drawer width. Positive numbers are converted to pixels; character values must be valid CSS lengths.
<code>open</code>	Whether the drawer is initially visible.
<code>resizable</code>	Whether the drawer can be resized on desktop.

Value

A configuration object for use with [chat_ui\(\)](#) or [page_chat\(\)](#).

See Also

[chat_ui\(\)](#) and [page_chat\(\)](#) accept this configuration through their `drawer` argument.

Other chat drawers: [chat_drawer_hide\(\)](#), [chat_drawer_show\(\)](#), [chat_drawer_toggle\(\)](#), [chat_drawer_update\(\)](#)

chat_drawer_hide	<i>Hide a chat drawer</i>
------------------	---------------------------

Description

Hide a chat drawer

Usage

```
chat_drawer_hide(id, session = shiny::getDefaultReactiveDomain())
```

Arguments

<code>id</code>	The ID of the chat element.
<code>session</code>	The Shiny session object.

Value

Invisibly, NULL.

See Also

[chat_drawer\(\)](#) to configure a drawer, and [chat_ui\(\)](#) or [page_chat\(\)](#) to display one.

Other chat drawers: [chat_drawer\(\)](#), [chat_drawer_show\(\)](#), [chat_drawer_toggle\(\)](#), [chat_drawer_update\(\)](#)

chat_drawer_show	Show a chat drawer
------------------	--------------------

Description

Shows a chat's drawer. Supplying content or title updates that field before the panel is shown. Omitted fields preserve their current value.

Usage

```
chat_drawer_show(  
  id,  
  content = NULL,  
  title = NULL,  
  session = shiny::getDefaultReactiveDomain()  
)
```

Arguments

id	The ID of the chat element.
content	Optional UI content for the drawer. Use an empty <code>htmltools::tagList()</code> to clear the content.
title	Optional drawer title. Use <code>""</code> to clear the title.
session	The Shiny session object.

Value

Invisibly, NULL.

See Also

[chat_drawer\(\)](#) to configure a drawer, and [chat_ui\(\)](#) or [page_chat\(\)](#) to display one.

Other chat drawers: [chat_drawer\(\)](#), [chat_drawer_hide\(\)](#), [chat_drawer_toggle\(\)](#), [chat_drawer_update\(\)](#)

chat_drawer_toggle	Toggle a chat drawer
--------------------	----------------------

Description

Toggle a chat drawer

Usage

```
chat_drawer_toggle(id, session = shiny::getDefaultReactiveDomain())
```

Arguments

id	The ID of the chat element.
session	The Shiny session object.

Value

Invisibly, NULL.

See Also

[chat_drawer\(\)](#) to configure a drawer, and [chat_ui\(\)](#) or [page_chat\(\)](#) to display one.

Other chat drawers: [chat_drawer\(\)](#), [chat_drawer_hide\(\)](#), [chat_drawer_show\(\)](#), [chat_drawer_update\(\)](#)

chat_drawer_update	<i>Update a chat drawer</i>
--------------------	-----------------------------

Description

Updates a chat's drawer content or title without changing its visibility. Omitted fields preserve their current value. Use an empty [htmltools::tagList\(\)](#) to clear content or "" to clear the title.

Usage

```
chat_drawer_update(  
  id,  
  content = NULL,  
  title = NULL,  
  session = shiny::getDefaultReactiveDomain()  
)
```

Arguments

id	The ID of the chat element.
content	Optional UI content for the drawer.
title	Optional drawer title.
session	The Shiny session object.

Value

Invisibly, NULL.

See Also

[chat_drawer\(\)](#) to configure a drawer, and [chat_ui\(\)](#) or [page_chat\(\)](#) to display one.

Other chat drawers: [chat_drawer\(\)](#), [chat_drawer_hide\(\)](#), [chat_drawer_show\(\)](#), [chat_drawer_toggle\(\)](#)

chat_enable_history *Enable conversation history for a chat*

Description

Enable conversation history for a chat

Usage

```
chat_enable_history(
  id,
  client,
  ...,
  on_save = NULL,
  on_restore = NULL,
  options = history_options(),
  restore_ui = TRUE,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

id	The chat element ID.
client	An <code>ellmer::Chat</code> object.
...	Reserved for future use.
on_save	An optional function(values) called whenever the active conversation is saved. Receives a named list; add any per-conversation state you want to persist and return the modified list. Fired on each LLM response and when the user switches conversations. Multiple callbacks may be registered; they are called in registration order.
on_restore	An optional function(values) called when a conversation is loaded — after it becomes active, on page-load restore and on in-session switches. Use it to sync auxiliary UI state (tabs, model selectors, etc.) to match the restored conversation. Call the appropriate <code>updateXxx()</code> functions here. Receives the values list captured by <code>on_save</code> . Multiple callbacks may be registered; they are called in registration order. In <code>restore_mode = "bookmark"</code> , this callback also runs while Shiny restores native bookmarked inputs.
options	A <code>history_options()</code> object controlling storage, identity, titling, and restore behaviour.
restore_ui	Whether to render the active conversation into the chat UI and fire <code>on_restore</code> on registration. Default is <code>TRUE</code> . Set to <code>FALSE</code> when re-registering history after a client swap (where the UI already reflects the conversation).
session	The Shiny session.

Value

Invisibly, a function that cancels all history registrations.

chat_get_greeting	<i>Get the current greeting content</i>
-------------------	---

Description

Get the current greeting content

Usage

```
chat_get_greeting(id, session = getDefaultReactiveDomain())
```

Arguments

id	The ID of the chat element
session	The Shiny session object

Value

A character string with the current greeting content, or NULL if no greeting is set or has been cleared.

chat_greeting	<i>Create a greeting for a chat UI</i>
---------------	--

Description

Creates a greeting object for use with [chat_ui\(\)](#) or [chat_set_greeting\(\)](#). A greeting is displayed when the chat first loads and is dismissed when the user sends their first message.

Usage

```
chat_greeting(
  content,
  ...,
  persistent = FALSE,
  dismissible = lifecycle::deprecated()
)
```

Arguments

content	The greeting content. Can be: • A string, interpreted as markdown. • An <code>htmltools::HTML()</code> object, rendered as raw HTML. • An <code>htmltools</code> tag or <code>htmltools::tagList()</code>, including Shiny inputs/outputs. • A generator or promise (only valid when used with <code>chat_set_greeting()</code>).
...	These dots are for future extensions and must be empty.
persistent	Whether the greeting persists after the user sends a message. Defaults to <code>FALSE</code> , meaning the greeting is dismissed when the user sends their first message.
dismissible	[Deprecated] Renamed to <code>persistent</code> with an inverted value. <code>dismissible = FALSE</code> is equivalent to <code>persistent = TRUE</code> .

Value

An S3 object of class "chat_greeting".

Patterns

Persistent greeting (stays visible after the user sends a message):

```
chat_greeting("Please read our [terms of service](https://example.com).", persistent = TRUE)
```

Greeting with suggestion cards (clickable chips that fill the input):

```
chat_greeting(paste(
  "## Welcome!\n\n",
  "Try one of these:\n\n",
  '<span class="suggestion">Summarize my data</span>\n',
  '<span class="suggestion">Create a plot</span>\n',
  '<span class="suggestion">Explain this code</span>'
))
```

Greeting with HTML tags (Shiny inputs/outputs):

```
chat_greeting(htmltools::tagList(
  htmltools::h2("Welcome!"),
  shiny::selectInput("model", "Choose a model:", c("gpt-4o", "claude-3"))
))
```

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  chat_ui(
```

```

      "chat",
      greeting = chat_greeting("## Welcome!\n\nHow can I help you today?")
    )
  )

  server <- function(input, output, session) {
    observeEvent(input$chat_user_input, {
      response <- paste0("You said: ", input$chat_user_input)
      chat_append("chat", response)
    })
  }

  shinyApp(ui, server)

```

chat_nav_panel

Create a page-chat navigation panel

Description

Creates a secondary page for `page_chat()` in the same style as `bslib::nav_panel()` and `bslib::page_navbar()`. When users navigate to the panel, the chat remains mounted on the home page so its conversation and UI state persist.

Usage

```

chat_nav_panel(
  title,
  ...,
  value = NULL,
  icon = NULL,
  sidebar = FALSE,
  toolbar = NULL,
  content_width = "min(680px, 100%)"
)

```

Arguments

title	The panel title.
...	UI content to display when the panel is active.
value	An optional unique navigation value. Defaults to title. The value <code>"__home__"</code> is reserved for the main chat page.
icon	An optional icon to display with the title.
sidebar	Whether to use the default sidebar (TRUE), no page-specific sidebar (FALSE), or a <code>chat_sidebar()</code> or <code>bslib::sidebar()</code> configuration. A <code>chat_sidebar()</code> with <code>history = NULL</code> defaults to FALSE here.

toolbar	NULL (the default) for no page-scoped toolbar, or UI content for a page-specific toolbar. Use <code>bslib::toolbar()</code> to group toolbar controls.
content_width	Maximum panel-content width. Content is centered and receives responsive in-line padding. Use exactly "100%", "100vw", or "100dvw" for full-bleed content without component-provided padding.

Value

A configuration object for use with `page_chat()`.

chat_restore	<i>Add Shiny bookmarking for shinychat</i>
--------------	--

Description

Adds Shiny bookmarking hooks to save and restore the **ellmer** chat client. Also restores chat messages from the history in the client.

If either `bookmark_on_input` or `bookmark_on_response` is TRUE, the Shiny App's bookmark will be automatically updated without showing a modal to the user.

Note: The client's chat state and the greeting content are both saved/restored automatically. If the client's state doesn't properly capture the chat's UI (i.e., a transformation is applied in-between receiving and displaying the message), you may need to implement your own `session$onRestore()` (and possibly `session$onBookmark`) handler to restore any additional state.

To avoid restoring chat history from the client, you can ensure that the history is empty by calling `client$set_turns(list())` before passing the client to `chat_restore()`.

`chat_restore()` bookmarks the whole session and doesn't know about multiple conversations. If you need per-conversation history (the chat history drawer, switching between saved conversations), use `chat_enable_history()` with `history_options(restore_mode = "bookmark")` instead — it replaces `chat_restore()`'s job for history-aware apps. The two are mutually exclusive; `chat_app()` picks one or the other based on whether history is set.

Usage

```
chat_restore(
  id,
  client,
  ...,
  bookmark_on_input = TRUE,
  bookmark_on_response = TRUE,
  restore_ui = TRUE,
  session = getDefaultReactiveDomain()
)
```

Arguments

<code>id</code>	The ID of the chat element
<code>client</code>	The ellmer LLM chat client.
<code>...</code>	Used for future parameter expansion.
<code>bookmark_on_input</code>	A logical value determines if the bookmark should be updated when the user submits a message. Default is TRUE.
<code>bookmark_on_response</code>	A logical value determines if the bookmark should be updated when the response stream completes. Default is TRUE.
<code>restore_ui</code>	Whether to render the client's existing turns into the chat UI on registration. Default is TRUE. Set to FALSE when re-registering bookmarks after a client swap (where the UI already reflects the conversation).
<code>session</code>	The Shiny session object

Value

Invisibly returns a function that, when called, cancels all bookmark registrations made by this call. This is useful when swapping the chat client: cancel the previous bookmarks, then call `chat_restore()` again with the new client.

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- function(request) {
  page_fillable(
    chat_ui("chat", fill = TRUE)
  )
}

server <- function(input, output, session) {
  chat_client <- ellmer::chat_ollama(
    system_prompt = "Important: Always respond in a limerick",
    model = "qwen2.5-coder:1.5b",
    echo = TRUE
  )
  # Update bookmark to chat on user submission and completed response
  chat_restore("chat", chat_client)

  observeEvent(input$chat_user_input, {
    stream <- chat_client$stream_async(input$chat_user_input)
    chat_append("chat", stream)
  })
}

# Enable bookmarking!
```

```
shinyApp(ui, server, enableBookmarking = "server")
```

chat_set_greeting	<i>Set the greeting for a chat UI</i>
-------------------	---------------------------------------

Description

Sets or clears the greeting displayed in an existing `chat_ui()`. The greeting is shown when the chat first loads and is dismissed when the user sends their first message.

Call `chat_set_greeting()` from the server to display a dynamic or streaming greeting. This is typically used inside an `shiny::observeEvent()` watching `input$<id>_greeting_requested` – an event that fires when the chat is visible, has no messages, and has no greeting set. See the **Greeting** section in `chat_ui()` for details on `greeting_requested`.

If the greeting has already been dismissed, calling this function updates the greeting content but does not make it visible again. To show a new greeting after dismissal, first clear the chat with `chat_clear(id, greeting = TRUE)`.

Pass `NULL` to clear the current greeting entirely.

Usage

```
chat_set_greeting(id, greeting, session = getDefaultReactiveDomain())
```

Arguments

<code>id</code>	The ID of the chat element
<code>greeting</code>	The greeting to display. Can be: • <code>NULL</code> to clear the current greeting. • A plain string or <code>htmltools::HTML()</code> object (auto-wrapped with default options). • A <code>chat_greeting()</code> object.
<code>session</code>	The Shiny session object

Value

Returns `invisible(NULL)` for static and `NULL` greetings, or a promise for streaming greetings (resolves when streaming is complete).

Examples

```
library(shiny)
library(bslib)
library(shinychat)

# Static greeting set from the server
ui <- page_fillable(chat_ui("chat"))
```

```

server <- function(input, output, session) {
  chat_set_greeting("chat", "## Welcome!\n\nHow can I help you today?")

  observeEvent(input$chat_user_input, {
    chat_append("chat", paste0("You said: ", input$chat_user_input))
  })
}

shinyApp(ui, server)

library(shiny)
library(bslib)
library(shinychat)
library(coro)

# Streaming greeting generated by a function
greeting_generator <- async_generator(function() {
  for (chunk in strsplit("## Hello!\n\nHow can I help you?", "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(chat_ui("chat"))

server <- function(input, output, session) {
  chat_set_greeting("chat", chat_greeting(greeting_generator()))

  observeEvent(input$chat_user_input, {
    chat_append("chat", paste0("You said: ", input$chat_user_input))
  })
}

shinyApp(ui, server)

library(shiny)
library(bslib)
library(shinychat)

# LLM-generated greeting using greeting_requested
ui <- page_fillable(chat_ui("chat"))

server <- function(input, output, session) {
  chat_client <- ellmer::chat_openai(model = "gpt-4o")

  observeEvent(input$chat_greeting_requested, {
    stream <- chat_client$stream_async(
      "Generate a short, friendly welcome message."
    )
    chat_set_greeting("chat", chat_greeting(stream))
  })
}

```

```

    })

    observeEvent(input$chat_user_input, {
      stream <- chat_client$stream_async(input$chat_user_input)
      chat_append("chat", stream)
    })
  }

shinyApp(ui, server)

library(shiny)
library(bslib)
library(shinychat)

# Regenerate pattern: chat_clear(greeting = TRUE) triggers greeting_requested
ui <- page_fillable(
  chat_ui("chat"),
  actionButton("regenerate", "New greeting")
)

server <- function(input, output, session) {
  observeEvent(input$chat_greeting_requested, {
    chat_set_greeting(
      "chat",
      paste("## Welcome!\n\nGenerated at", Sys.time())
    )
  })

  observeEvent(input$regenerate, {
    chat_clear("chat", greeting = TRUE)
  })

  observeEvent(input$chat_user_input, {
    chat_append("chat", paste0("You said: ", input$chat_user_input))
  })
}

shinyApp(ui, server)

```

chat_sidebar

Create a chat sidebar configuration

Description

Configures sidebar content for the home view or a `chat_nav_panel()` in `page_chat()`. A page-chat sidebar behaves like a compact `bslib::sidebar()` beside the chat and can include the chat's conversation history selector.

Usage

```
chat_sidebar(..., history = NULL, width = 280, open = "auto", resizable = TRUE)
```

Arguments

<code>...</code>	UI content to display in the sidebar.
<code>history</code>	Whether to display the chat history selector in the sidebar. When <code>NULL</code> , <code>page_chat()</code> defaults to <code>TRUE</code> and <code>chat_nav_panel()</code> defaults to <code>FALSE</code> .
<code>width</code>	The initial sidebar width. Positive numbers are converted to pixels; character values must be valid CSS lengths.
<code>open</code>	The initial sidebar state. One of "auto", "open", "closed", or "always". Logical values are aliases for "open" and "closed".
<code>resizable</code>	Whether the sidebar can be resized on desktop.

Value

A configuration object for use with `page_chat()` or `chat_nav_panel()`.

Examples

```
ui <- page_chat(
  "Assistant",
  sidebar = chat_sidebar(
    shiny::tags$p("Workspace"),
    history = TRUE,
    open = "open"
  )
)
```

chat_ui

Create a chat UI element

Description

Inserts a chat UI element into a Shiny UI, which includes a scrollable section for displaying chat messages, and an input field for the user to enter new messages.

To respond to user input, listen for `input$ID_user_input` (for example, if `id="my_chat"`, user input will be at `input$my_chat_user_input`), and use `chat_append()` to append messages to the chat.

Usage

```
chat_ui(
  id,
  ...,
  greeting = NULL,
  placeholder = "Enter a message...",
  drawer = TRUE,
  footer = NULL,
  toolbar_input = NULL,
  show_history = TRUE,
  show_thinking_after_s = 0,
  width = "min(clamp(680px, 50vw, 760px), 100%)",
  height = "auto",
  fill = TRUE,
  icon_assistant = NULL,
  icon_send = NULL,
  enable_cancel = NULL,
  submit_key = c("enter", "enter+modifier"),
  allow_attachments = NULL,
  tool_grouping = c("tool", "none", "all"),
  messages = NULL
)
```

Arguments

<code>id</code>	The ID of the chat element
<code>...</code>	Extra HTML attributes to include on the chat element
<code>greeting</code>	An optional greeting to display when the chat first loads. Can be a chat_greeting() object, or a plain string (which is auto-wrapped with default options). The greeting is dismissed when the user sends their first message. For example: <code>greeting = chat_greeting("## Hello!\n\nHow can I help you today?")</code>
<code>placeholder</code>	The placeholder text for the chat's user input field
<code>drawer</code>	Whether to enable the drawer. <code>TRUE</code> (the default) enables an initially hidden panel with default options, <code>FALSE</code> omits it, and chat_drawer() supplies its initial configuration.
<code>footer</code>	Optional HTML content to display in a bottom-pinned, full-width chat region. This can be any HTML content (tags, tag lists, or character strings). Useful for adding disclaimers, attribution, or other information. The footer text is styled slightly smaller and lighter than body text by default. Customize with CSS properties <code>--shiny-chat-footer-font-size</code> and <code>--shiny-chat-footer-color</code> on the chat container or footer element.
<code>toolbar_input</code>	Optional HTML content to display directly below the chat input. Use bslib::toolbar() to group toolbar controls.
<code>show_history</code>	Whether to show the built-in history selector. Defaults to <code>TRUE</code> ; setting it to <code>FALSE</code> only hides its presentation.

show_thinking_after_s	The minimum number of seconds a contiguous thinking block must run before it is displayed. Defaults to 0, which displays thinking immediately. Positive values hide shorter blocks; negative values always hide thinking. Values must not exceed 60 seconds.
width	The CSS width of the chat element
height	The CSS height of the chat element
fill	Whether the chat element should try to vertically fill its container, if the container is fillable
icon_assistant	The icon to use for the assistant chat messages. Can be HTML or a tag in the form of <code>htmltools::HTML()</code> or <code>htmltools::tags()</code> . NULL (the default) or FALSE omits the assistant icon entirely. Pass TRUE to use the built-in robot icon (individual messages can still opt in to a different icon via the <code>icon</code> argument of <code>chat_append()</code>).
icon_send	The icon to use for the chat input's ready-state submit button. Can be HTML or a tag in the form of <code>htmltools::HTML()</code> or <code>htmltools::tags()</code> . If NULL (the default) or FALSE, a default arrow icon is used. The button provides a filled circular surface (state-colored background, white icon); the supplied icon replaces only the glyph inside it. See the "Customizing the send button" section below for styling patterns.
enable_cancel	Whether to show a stop button during streaming that allows the user to cancel the in-progress response. When using <code>chat_server()</code> , cancellation is wired up automatically and this defaults to NULL (let the server decide). For manual usage without <code>chat_server()</code> , set TRUE or FALSE explicitly and observe <code>input\$<id>_cancel</code> to handle cancellation (e.g., by calling <code>ctrl\$cancel()</code> on an ellmer <code>stream_controller()</code>).
submit_key	Controls which key combination submits the chat message. "enter" (the default): Enter submits, Shift+Enter adds a newline. "enter+modifier": Ctrl+Enter (Cmd+Enter on Mac) submits, plain Enter adds a newline.
allow_attachments	Controls the file-attachment affordance (an attach button, plus clipboard paste and drag-and-drop) in the chat input. NULL (default) defers to <code>chat_server()</code>, which enables attachments automatically. Pass TRUE to accept all supported types (PNG, JPEG, GIF, WebP, PDF, and common text/code files such as Markdown, plain text, CSV, JSON, and source files), FALSE to disable, or a character vector of MIME types to restrict what is accepted (each must be one of the supported types). The shape of <code>input\$<id>_user_input</code> is determined by this argument, so it is predictable for a given app. When attachments are disabled (the default), it is the typed text as a character string, exactly as before. When attachments are enabled, it is always a list of ellmer <code>ellmer::Content</code> objects (the typed text, if any, followed by one content object per attachment) - a list even when no files were attached. Splice the list into a chat method's ... with <code>!!!</code>, e.g. <code>client\$stream_async(!!!input\$<id>_user_input)</code>. (No <code>rlang::inject()</code> is needed: ellmer's chat methods collect ... with dynamic dots.)

The maximum combined size of all attachments in a single message is controlled globally by the `SHINYCHAT_MAX_ATTACHMENT_SIZE` environment variable (a raw byte count; defaults to approximately 30 MB). Files that would push the total over this cap are rejected in the browser with a notice.

tool_grouping	Controls how tool calls are grouped together in the compact activity rows: • "tool" (default): calls to the <i>same</i> tool within a turn's contiguous tool loop are grouped into one activity row. This groups by tool name across the whole loop, not just consecutive calls – e.g. calls to tools X, Y, Z, X, Y (in that order) are grouped into X (2 calls), Y (2 calls), and Z (1 call). • "all": every tool call within a contiguous tool loop is summarized in one activity row, regardless of tool name. • "none": each tool call is shown in its own activity row. Its request and result remain available by drilling into that row; this does not restore an always-visible card stack. Prose or thinking between tool calls starts a new tool loop, so calls on opposite sides of either boundary never group together. Individual tools can override "tool" or "all" via a top-level grouping tool annotation, e.g. <code>ellmer::tool(..., annotations = ellmer::tool_annotations(grouping = "all"))</code>. <code>tool_grouping = "none"</code> takes precedence over every annotation and disables grouping for the whole chat.
messages	Deprecated. A list of messages to prepopulate the chat with. Startup messages can't be recorded by the conversation-history feature. Use <code>greeting</code> for a startup message, <code>chat_append()</code> to replay messages from the server, or set <code>history = FALSE</code> in <code>chat_server()</code> if you're managing conversation state yourself. Each message can be one of the following: • A string, which is interpreted as markdown and rendered to HTML on the client. – To prevent interpreting as markdown, mark the string as <code>htmltools::HTML()</code>. • A UI element. – This includes <code>htmltools::tagList()</code>, which takes UI elements (including strings) as children. Strings inside a <code>tagList</code> are literal text (HTML-escaped), not markdown. Use <code>htmltools::HTML()</code> for trusted raw HTML strings. • A named list of content and role. The content can contain content as described above, and the role can be "assistant" or "user". • Advanced: a <code>list()</code> mixing bare strings and UI elements interleaves markdown and HTML in one message, in order. This API is provisional and may change in a future release.

Value

A Shiny tag object, suitable for inclusion in a Shiny UI

Pairing with `chat_server()`

`chat_ui(id)` and `chat_server(id, client)` pair by matching `id`. This works the same way at the top level of an app and inside your own Shiny module — `chat_server()` is not itself a module, so `noNS(id, "chat")` wrapping is required:

```
# Top-level app, no module
ui <- page_fillable(chat_ui("chat"))
server <- function(input, output, session) {
  chat_server("chat", client)
}

# Inside your own module: pass the same literal id to both, and call
# chat_server() from inside moduleServer() so it inherits the module's
# already-namespaced `session`
mod_ui <- function(id) {
  ns <- NS(id)
  chat_ui(ns("chat"))
}
mod_server <- function(id, client) {
  moduleServer(id, function(input, output, session) {
    chat_server("chat", client)
  })
}
```

Greeting

A greeting is an optional welcome message shown before any conversation messages. It is automatically dismissed when the user sends their first message (unless created with `persistent = TRUE`).

Static greeting. Pass a string or `chat_greeting()` to the greeting parameter:

```
chat_ui("chat", greeting = "## Hello!\n\nHow can I help you today?")
```

Dynamic greeting from the server. Leave greeting unset and use `chat_set_greeting()` from your server function. This is useful when the greeting depends on session state or is generated by a model.

greeting_requested input. When the chat is visible on the page, has no messages, and has no greeting set, Shiny fires `input$<id>_greeting_requested` (e.g. `input$chat_greeting_requested` for `chat_ui("chat")`). The value is an event counter suitable for `shiny::observeEvent()`. Use it to trigger server-side greeting generation:

```
observeEvent(input$chat_greeting_requested, {
  stream <- chat_client$stream_async("Generate a short welcome message.")
  chat_set_greeting("chat", chat_greeting(stream))
})
```

This input fires when the chat component is first viewed on the page and empty, and again after `chat_clear()` (`greeting = TRUE`), enabling a regenerate pattern where clearing the greeting automatically triggers a fresh one.

greeting_dismissed input. When the user dismisses the greeting, `input$<id>_greeting_dismissed` fires with a `Date.now()` timestamp. If the greeting is later cleared after being dismissed, the input resets to `NULL`. If you use `chat_server()`, you can access the `greeting_dismissed` reactive from the returned value instead of the raw namespaced input string.

Thinking display

When a model produces reasoning or "thinking" tokens, shinychat renders them in a collapsible panel above the response. The panel shows a live stream of the model's reasoning while it thinks, then auto-collapses when the response begins.

Thinking display works automatically with any model that supports it. Two paths are supported:

1. **ellmer's ContentThinking objects.** Models that provide a structured thinking API (e.g., Claude with extended thinking) emit ContentThinking objects when you stream with `stream = "content"`. shinychat detects these and routes them to the thinking panel. This is what `chat_append()` uses internally when you pass it an ellmer content stream.
2. **Raw <thinking> tags.** Many open-source and local models (DeepSeek, QwQ, Qwen, etc.) emit `<thinking>...</thinking>` tags directly in their markdown output. shinychat detects these tags during streaming and renders the enclosed text in the thinking panel with no extra configuration.

Topic labels:

You can optionally get labeled sub-sections within the thinking panel by asking the model to emit `<topic>...</topic>` tags in its reasoning. These are extracted and rendered as section headings inside the thinking display, and the current topic appears in the collapsed header as a live status.

To use topic labels, add something like this to your system prompt:

```
When thinking through a problem, wrap brief topic labels in <topic> tags
to indicate what you're currently reasoning about. For example:
<topic>parsing the input</topic>
```

Topic labels are entirely optional. Without them, the thinking panel still works – it just won't have sub-section headings.

Set `show_thinking_after_s` in `chat_ui()` to delay the thinking panel until a contiguous thinking block has run for a minimum number of seconds. The default, `0`, displays thinking immediately; a negative value always hides it. Values greater than 60 seconds are not supported. Durations are measured in the browser while streaming, so preloaded or restored thinking is displayed only with the default.

Customizing the send button

The send button is a filled circle (24px by default) whose background color reflects the current state (primary when ready, gray when empty/disabled, danger when cancelling) with a white icon (22px by default) centered inside. The `icon_send` parameter swaps the ready-state icon without changing the button's surface.

Custom icon. Pass an SVG from `bsicons::bs_icon()` or `fontawesome::fa()`. The button provides the surface, so a bare glyph gets the same filled-circle treatment as the default arrow:

```
chat_ui("chat", icon_send = bsicons::bs_icon("send-fill"))
```

Icon with text. Use `htmltools::tagList()` to pass an icon and a text label as siblings (not wrapped in a `<span>`) so they lay out side by side, with a Bootstrap margin utility for spacing. Then override the button to size to its content instead of the default fixed circle:

```
chat_ui("chat",
  icon_send = tagList(
    bsicons::bs_icon("airplane-fill"),
    span("Send", class = "ms-2")
  )
)

:root .shiny-chat-btn-send {
  width: auto;
  height: auto;
  padding: 4px 10px;
  border-radius: 6px;
}
```

Per-state color overrides. Each state's color can be set independently via CSS variables on the chat container or any ancestor. These are only read by the component (never set on the button), so inline styles inherit cleanly:

```
#chat {
  --shiny-chat-btn-send-color-cancel: #abc123;
}
```

Ghost (outline) style. Make the button transparent at rest with the state color on the icon and border, filling on hover. Target the button element (not an ancestor) because the internal `--_btn-send-state-color` variable resolves on the button itself:

```
:root .shiny-chat-btn-send {
  --shiny-chat-btn-send-bg: transparent;
  --shiny-chat-btn-send-color: var(--_btn-send-state-color);
  --shiny-chat-btn-send-border: 1px solid var(--_btn-send-state-color);
  --shiny-chat-btn-send-color-hover: #fff;
  --shiny-chat-btn-send-bg-hover: var(--_btn-send-state-color);
}
```

Key CSS variables:

- `--shiny-chat-btn-send-size` — Button width and height (default 24px)
- `--shiny-chat-input-icon-size` — Icon size, shared with the attach button (default 22px)
- `--shiny-chat-btn-send-bg` — Button background (default: state color)
- `--shiny-chat-btn-send-color` — Icon color (default: #fff)
- `--shiny-chat-btn-send-border` — Button border (default: none)
- `--shiny-chat-btn-send-color-ready` — Override ready/pending color (default: `--bs-primary`)
- `--shiny-chat-btn-send-color-empty` — Override empty/disabled color (default: `--bs-gray-500`)
- `--shiny-chat-btn-send-color-cancel` — Override cancel/cancelling color (default: `--bs-danger`)

Examples

```

library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  chat_ui("chat", fill = TRUE)
)

server <- function(input, output, session) {
  observeEvent(input$chat_user_input, {
    # In a real app, this would call out to a chat client or API,
    # perhaps using the 'ellmer' package.
    response <- paste0(
      "You said:\n\n",
      "<blockquote>",
      htmltools::htmlEscape(input$chat_user_input),
      "</blockquote>"
    )
    chat_append("chat", response)
  })
}

shinyApp(ui, server)

```

chat_ui_history	<i>Create a chat history selector</i>
-----------------	---------------------------------------

Description

Create a chat history selector

Usage

```
chat_ui_history(id, ...)
```

Arguments

id	The ID of the associated chat.
...	Named HTML attributes to apply to the selector.

Value

A Shiny tag object.

ContentSlashCommand	<i>Slash command content</i>
---------------------	------------------------------

Description

An [ellmer::ContentText](#) subclass that preserves the original slash command entered by the user. When the chat UI is restored from a bookmark (or pre-existing turns), the original /command user_text is shown in the UI instead of the (possibly transformed) text that was sent to the LLM.

Usage

```
ContentSlashCommand(
  text = stop("Required"),
  command = character(0),
  user_text = ""
)
```

Arguments

text	The text sent to the LLM. When constructed by the chat module, this defaults to a descriptive string like "The user entered the /greet slash command with arguments: world". Set it in the handler to control what the LLM actually sees.
command	The slash command name (without the leading /).
user_text	The text the user typed after the command name. Defaults to "".

Value

A ContentSlashCommand object.

How it works

Slash command handlers that accept an argument receive a ContentSlashCommand object rather than a plain string. The object has three properties:

- `command`: the command name (e.g., "greet").
- `user_text`: the text the user typed after the command (e.g., "world").
- `text`: the text that will be sent to the LLM. This starts as a descriptive string like "The user entered the /greet slash command with arguments: world". Set it to whatever text the LLM should actually see.

Because ContentSlashCommand extends [ellmer::ContentText](#), it works anywhere a ContentText does – including `client$stream()`, `client$chat()`, etc. LLM providers read the `text` property (inherited behavior), while `contents_shinychat()` reconstructs the original /command user_text for display in the chat UI.

Bookmark serialization via [ellmer::contents_record\(\)](#) / [ellmer::contents_replay\(\)](#) is automatic.

Example

```
chat$slash_command("greet", "Greet someone", function(content) {
  content@text <- paste("Say hello to", content@user_text)
  stream <- client$stream(content)
  chat_append("chat", stream)
})
```

The LLM sees "Say hello to world", but on restore the chat UI shows /greet world.

See Also

[chat_server\(\)](#) for registering slash commands via the `slash_command()` method on the returned object.

contents_shinychat	<i>Format ellmer content for shinychat</i>
--------------------	--

Description

Format ellmer content for shinychat

Usage

```
contents_shinychat(content)
```

Arguments

content An `ellmer::Content` object.

Value

Returns text, HTML, or web component tags formatted for use in `chat_ui()`.

Extending contents_shinychat()

You can extend `contents_shinychat()` to handle custom content types in your application. `contents_shinychat()` is an [S7 generic](#). If you haven't worked with S7 before, you can learn more about S7 classes, generics and methods in the [S7 documentation](#).

For most tool-result customization, use `tool_result_display()` in the result's `extra = list(display = ...)`. It keeps shinychat's compact activity row and drill-down card while letting you set a title, label, result preview, and rich card content. The [Tool Calling UI article](#) describes that recommended path.

We'll work through a short example creating a custom display for the results of a tool that gets local weather forecasts. We first need to create a custom class that extends `ellmer::ContentToolResult`.

```
library(ellmer)

WeatherToolResult <- S7::new_class(
  "WeatherToolResult",
  parent = ContentToolResult,
  properties = list(
    location_name = S7::class_character
  )
)
```

Next, we'll create a simple `ellmer::tool()` that gets the weather forecast for a location and returns our custom `WeatherToolResult` class. The custom class works just like a regular `ContentToolResult`, but it has an additional `location_name` property.

```
get_weather_forecast <- tool(
  function(lat, lon, location_name) {
    WeatherToolResult(
      weathR::point_tomorrow(lat, lon, short = FALSE),
      location_name = location_name
    )
  },
  name = "get_weather_forecast",
  description = "Get the weather forecast for a location.",
  arguments = list(
    lat = type_number("Latitude"),
    lon = type_number("Longitude"),
    location_name = type_string("Name of the location for display to the user")
  )
)
```

Finally, define the external generic and implement a method for your custom class:

```
contents_shinychat <- S7::new_external_generic(
  package = "shinychat",
  name = "contents_shinychat",
  dispatch_args = "contents"
)

S7::method(contents_shinychat, WeatherToolResult) <- function(content) {
  # Your custom rendering logic here
}
```

Use `S7::super()` when you want to extend shinychat's default card. The resulting output still participates in the normal compact activity row and drill-down card:

```
S7::method(contents_shinychat, WeatherToolResult) <- function(content) {
  # Call the super method for ContentToolResult to get shinychat's defaults
  res <- contents_shinychat(S7::super(content, ContentToolResult))
}
```

```

# Then update the result object with more specific content
# In this case, we render the tool result dataframe as a {gt} table...
res$value <- gt::as_raw_html(gt::gt(content@value))
res$value_type <- "html"
# ...and update the tool result title to include the location name
res$title <- paste("Got weather forecast for", content@location_name)
res$label <- content@location_name
res$value_preview <- paste(nrow(content@value), "hourly readings")

res
}

```

Alternatively, return arbitrary HTML or Shiny UI directly from the method to replace the default card completely. While the tool runs, shinychat still shows its activity row. When the custom result settles, shinychat renders that UI as standalone output and removes the call from the activity row.

This extension point is for fully custom standalone output. To customize the default card, use [tool_result_display\(\)](#) instead of constructing a generic display list yourself.

ConversationStore

Abstract base class for conversation storage backends

Description

Subclass this to plug a custom persistence backend into [chat_enable_history\(\)](#) via `history_options(store = )`. All methods are partitioned by a `conversation_partition()` (chat id + owner scope); implementations should not need to know about users, sessions, or Shiny beyond that.

A conversation record is a list with fields `schema_version`, `id`, `title`, `title_source` ("llm", "user", or NULL), `response_count`, `created_at`, `updated_at` (ISO 8601 strings), `client_info`, `nodes` (a named list of turn nodes forming the conversation tree), `current_leaf` (id of the most recent node, or NULL), `values` (the app state dict captured by `on_save`), and `bookmark_state_id`. A conversation meta list is the lightweight summary returned by `list()`: `id`, `title`, `created_at`, `updated_at`, and `size_bytes` (the backend's storage footprint for that conversation, e.g. on-disk bytes).

`schema_version` compatibility is enforced by the framework's HistoryController, not by this class – it calls `check_schema_version()` on every record it reads from a store and every record it is about to write, so a custom store doesn't need to check it itself.

Methods

Public methods:

- [ConversationStore\\$list\(\)](#)
- [ConversationStore\\$get\(\)](#)
- [ConversationStore\\$put\(\)](#)
- [ConversationStore\\$delete\(\)](#)

- `ConversationStore$search()`
- `ConversationStore$total_size()`
- `ConversationStore$clone()`

`ConversationStore$list()`: Must be implemented by subclasses. All conversations in partition, newest-first by `created_at`.

Usage:

`ConversationStore$list(partition)`

Arguments:

`partition` A `conversation_partition()`.

Returns: A list of conversation meta lists.

`ConversationStore$get()`: Must be implemented by subclasses. The full conversation record for `id` in partition.

Usage:

`ConversationStore$get(partition, id)`

Arguments:

`partition` A `conversation_partition()`.

`id` A conversation id, as found in the `id` field of a conversation meta list.

Returns: The conversation record, or NULL if missing.

`ConversationStore$put()`: Must be implemented by subclasses. Upsert record into partition. A rename is just mutating `record$title` and calling `put()` again.

Usage:

`ConversationStore$put(partition, record)`

Arguments:

`partition` A `conversation_partition()`.

`record` A conversation record, in the same shape returned by `get()`.

Returns: NULL, invisibly.

`ConversationStore$delete()`: Must be implemented by subclasses. Remove the conversation `id` from partition. Missing ids are a no-op.

Usage:

`ConversationStore$delete(partition, id)`

Arguments:

`partition` A `conversation_partition()`.

`id` A conversation id, as found in the `id` field of a conversation meta list.

Returns: NULL, invisibly.

`ConversationStore$search()`: Case-insensitive substring match of query against title, over `list(partition)`. Backends don't need to override this unless they have a more efficient search path.

Usage:

ConversationStore\$search(partition, query)

Arguments:

partition A conversation_partition().

query A search string.

Returns: A list of conversation meta lists whose title matches query.

ConversationStore\$total_size(): Total bytes used by all conversations in partition, derived from list()'s per-record size_bytes. Backends don't need to override this unless they have a cheaper way to compute it.

Usage:

ConversationStore\$total_size(partition)

Arguments:

partition A conversation_partition().

Returns: The total size in bytes, as a double.

ConversationStore\$clone(): The objects of this class are cloneable with this method.

Usage:

ConversationStore\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

FileConversationStore *File-based conversation storage backend*

Description

Uses temporary records and journal rollback to protect against ordinary I/O failures, but does not fsync files or directories. This store also does not coordinate concurrent access across processes; callers must serialize reads and writes for each conversation.

Super class

[ConversationStore](#) -> FileConversationStore

Methods**Public methods:**

- [FileConversationStore\\$new\(\)](#)
- [FileConversationStore\\$list\(\)](#)
- [FileConversationStore\\$get\(\)](#)
- [FileConversationStore\\$put\(\)](#)
- [FileConversationStore\\$delete\(\)](#)

- `FileConversationStore$clone()`

`FileConversationStore$new()`: Create a new file-based conversation store.

Usage:

```
FileConversationStore$new(dir = NULL)
```

Arguments:

`dir` Directory to store conversations under. Defaults to NULL, which resolves a redeploy-safe location at first use (see `resolve_history_dir()`).

`FileConversationStore$list()`: All conversations in partition, newest-first by `created_at`, read from one `record.json` per conversation directory on disk.

Usage:

```
FileConversationStore$list(partition)
```

Arguments:

`partition` A `conversation_partition()`.

Returns: A list of conversation meta lists.

`FileConversationStore$get()`: The full conversation record for `id` in `partition`, reassembled from `record.json`, `turns.jsonl`, and `ui.jsonl`.

Usage:

```
FileConversationStore$get(partition, id)
```

Arguments:

`partition` A `conversation_partition()`.

`id` A conversation id, as found in the `id` field of a conversation meta list.

Returns: The conversation record, or NULL if missing.

`FileConversationStore$put()`: Upsert record into `partition`, appending new turns and UI data to `turns.jsonl/ui.jsonl` and rewriting `record.json`.

Usage:

```
FileConversationStore$put(partition, record)
```

Arguments:

`partition` A `conversation_partition()`.

`record` A conversation record, in the same shape returned by `get()`.

Returns: NULL, invisibly.

`FileConversationStore$delete()`: Remove the conversation `id` from `partition` by deleting its directory. Missing ids are a no-op.

Usage:

```
FileConversationStore$delete(partition, id)
```

Arguments:

`partition` A `conversation_partition()`.

`id` A conversation id, as found in the `id` field of a conversation meta list.

Returns: NULL, invisibly.

FileConversationStore\$clone(): The objects of this class are cloneable with this method.

Usage:

FileConversationStore\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

history_options	Configure chat history options
-----------------	--------------------------------

Description

Configure chat history options

Usage

```
history_options(  
  restore_mode = c("browser", "url", "none", "bookmark"),  
  store = "auto",  
  scope = NULL,  
  title = "auto",  
  max_store_mb = 100  
)
```

Arguments

restore_mode	How a previous conversation is reloaded when the page opens. "browser" (the default) stores the active conversation ID in localStorage so it survives page reloads. "url" stores the ID as a plain ?shinychat_conversation_id=<id> query parameter so the active conversation is visible in the address bar and users can share or bookmark specific conversations; no server bookmarking configuration is required. "bookmark" participates in Shiny server bookmarking: after every LLM response a fresh server bookmark is minted and the address bar updates to ?_state_id=... Requires bookmarkStore = "server" in the Shiny app options. On in-session conversation switches, navigates to the target conversation's bookmark URL if one exists. "none" disables automatic restore entirely.
store	Storage backend: "auto" (default: memory in dev, file in production), "memory", "file", or a ConversationStore R6 instance. "auto" emits a once-per-session message announcing which backend was chosen; set options(shinychat.history_options.store_auto_notify = TRUE) to silence it.
scope	Storage namespace for conversations. A string, a function(session) returning a string, or NULL (default: uses session\$user if authenticated, otherwise a per-browser token). Pass a shared string to allow multiple users to share history — for example session\$groups[[1]] to scope by group, or a constant like "global" to share across all users.

title	Title generation strategy. "auto" (default) for LLM-generated titles, a function(recorded_turns) for custom titles, or NULL to skip LLM titling (the conversation keeps its initial timestamp-based name).
max_store_mb	Maximum total storage in megabytes per chat history partition. Oldest conversations are evicted when the limit is exceeded. Defaults to 100.

Value

A configuration object for use with `chat_enable_history()`.

markdown_stream	<i>Stream markdown content</i>
-----------------	--------------------------------

Description

Streams markdown content into a `output_markdown_stream()` UI element. A markdown stream can be useful for displaying generative AI responses (outside of a chat interface), streaming logs, or other use cases where chunks of content are generated over time.

Usage

```
markdown_stream(  
  id,  
  content_stream,  
  operation = c("replace", "append"),  
  session = getDefaultReactiveDomain()  
)
```

Arguments

id	The ID of the markdown stream to stream content to.
content_stream	A string generator (e.g., <code>coro::generator()</code> or <code>coro::async_generator()</code>), a string promise (e.g., <code>promises::promise()</code>), or a string promise generator. An item may also be a structured content block (a shinychat_block such as a web_search, web_search_results, or web_fetch block of the kind ellmer content normalization produces for <code>chat_append()</code>). Each block is sent as one complete, append-only structured block message. The client validates, groups, and renders it. Only the block types the stream client supports are accepted. html_block and the web_* family. Any other block type (e.g. a tool block, which the client would drop with a warning) raises an error.
operation	The operation to perform on the markdown stream. The default, "replace", will replace the current content with the new content stream. The other option, "append", will append the new content stream to the existing content.
session	The Shiny session object.

Value

A promise that resolves to the accumulated stream content as a single string. Structured blocks contribute nothing to the string.

Examples

```
library(shiny)
library(coro)
library(bslib)
library(shinychat)

# Define a generator that yields a random response
# (imagine this is a more sophisticated AI generator)
random_response_generator <- async_generator(function() {
  responses <- c(
    "What does that suggest to you?",
    "I see.",
    "I'm not sure I understand you fully.",
    "What do you think?",
    "Can you elaborate on that?",
    "Interesting question! Let's examine thi... **See more**"
  )

  await(async_sleep(1))
  for (chunk in strsplit(sample(responses, 1), "")[[1]]) {
    yield(chunk)
    await(async_sleep(0.02))
  }
})

ui <- page_fillable(
  actionButton("generate", "Generate response"),
  output_markdown_stream("stream")
)

server <- function(input, output, session) {
  observeEvent(input$generate, {
    markdown_stream("stream", random_response_generator())
  })
}

shinyApp(ui, server)
```

output_markdown_stream

Create a UI element for a markdown stream.

Description

Creates a UI element for a `markdown_stream()`. A markdown stream can be useful for displaying generative AI responses (outside of a chat interface), streaming logs, or other use cases where chunks of content are generated over time.

Usage

```
output_markdown_stream(
  id,
  ...,
  content = "",
  content_type = "markdown",
  auto_scroll = TRUE,
  width = "min(680px, 100%)",
  height = "auto"
)
```

Arguments

<code>id</code>	A unique identifier for this markdown stream.
<code>...</code>	Extra HTML attributes to include on the chat element
<code>content</code>	A string of content to display before any streaming occurs. When <code>content_type</code> is Markdown or HTML, it may also be UI element(s) such as input and output bindings.
<code>content_type</code>	The content type. Default is "markdown" (specifically, CommonMark). Supported content types include: * "markdown": markdown text, specifically CommonMark * "html": for rendering HTML content. * "text": for plain text.
<code>auto_scroll</code>	Whether to automatically scroll to the bottom of a scrollable container when new content is added. Default is True.
<code>width</code>	The width of the UI element.
<code>height</code>	The height of the UI element.

Value

A shiny tag object.

See Also

`markdown_stream()`

Description

`page_chat()` creates a fillable page containing one persistent `chat_ui()` home view, optional navigation pages, and a responsive app-menu sidebar.

Use `page_chat()` as the top-level page UI when the chat owns the full browser window. It owns the page layout, the single mounted chat, and the responsive app-menu controls. Use `chat_ui()` directly when the chat is embedded in an existing layout or alongside other top-level page content. For a standalone interactive chat application, use `chat_app()`, which composes `page_chat()` with `chat_server()`.

Usage

```
page_chat(  
  title,  
  icon = NULL,  
  ...,  
  id = "chat",  
  pages_navbar = NULL,  
  toolbar = NULL,  
  toolbar_global = bslib::toolbar(bslib::input_dark_mode()),  
  toolbar_input = NULL,  
  navbar_options = NULL,  
  sidebar = TRUE,  
  messages = NULL,  
  greeting = NULL,  
  placeholder = "Enter a message...",  
  width = "min(clamp(680px, 50vw, 760px), 100%)",  
  icon_assistant = NULL,  
  icon_send = NULL,  
  enable_cancel = NULL,  
  allow_attachments = NULL,  
  footer = NULL,  
  drawer = TRUE,  
  window_title = NA,  
  lang = NULL,  
  theme = page_chat_theme()  
)
```

Arguments

<code>title</code>	The display title. May be text or reactive/static UI.
<code>icon</code>	Optional UI displayed before title.

...	Named lower-frequency <code>chat_ui()</code> arguments and HTML attributes. <code>page_chat()</code> owns height, fill, and show_history; attempts to pass those arguments are rejected.
id	A non-empty string identifying the chat. The currently selected page is readable server-side as <code>input\$<id>_page</code> and settable via <code>bslib::nav_select()</code> . Use <code>bslib::nav_show()</code> and <code>bslib::nav_hide()</code> to reveal or hide nav controls. The reserved value <code>"__home__"</code> represents the main chat page.
pages_navbar	NULL or a list of <code>chat_nav_panel()</code> configurations and supported standard bslib navigation items. Standard content panels use the normal page-chat content width with no page-specific sidebar or toolbar. <code>bslib::nav_panel_hidden()</code> panels render their nav control hidden; use <code>bslib::nav_show()</code> to reveal it.
toolbar	Optional home-page-scoped UI displayed with the navigation controls. Use <code>bslib::toolbar()</code> to group toolbar controls. A panel's <code>chat_nav_panel(toolbar =)</code> replaces this scoped segment.
toolbar_global	Optional persistent UI displayed after the page-scoped toolbar in the navigation controls. Use <code>bslib::toolbar()</code> to group toolbar controls. Defaults to a toolbar containing <code>bslib::input_dark_mode()</code> ; use NULL to opt out. It remains mounted while secondary pages are selected and while controls move between desktop and mobile layouts.
toolbar_input	Optional UI displayed directly below the chat input. Use <code>bslib::toolbar()</code> to group toolbar controls. This is independent of the navigation toolbar.
navbar_options	Optional <code>bslib::navbar_options()</code> that styles the page title bar. Its bg, theme, underline, and HTML attributes are supported. position and collapsible are unsupported because <code>page_chat()</code> owns the full-window layout and responsive app menu.
sidebar	Whether to use the default history sidebar (TRUE), omit the default sidebar (FALSE), or use a <code>chat_sidebar()</code> or <code>bslib::sidebar()</code> configuration. A bslib sidebar supplies its child content, width, initial open state, and resizability; its history defaults to FALSE. A <code>chat_sidebar()</code> with history = NULL defaults to TRUE here.
messages, greeting, placeholder, width, icon_assistant, icon_send, enable_cancel, allow_attachments, footer, drawer	Common arguments passed to <code>chat_ui()</code> .
window_title	A static browser-window title. The default, NA, derives the window title from title when title is a scalar string. Use NULL to omit the window title.
lang	An optional non-empty document language string.
theme	A <code>bslib::bs_theme()</code> object. Defaults to <code>page_chat_theme()</code> . Supply <code>bslib::bs_theme()</code> directly to use another bslib preset or a completely custom Bootstrap theme.

Value

A fillable bslib page.

Migration from page_fillable()

Replace:

```
bslib::page_fillable(chat_ui("chat", fill = TRUE))
```

with:

```
page_chat("Assistant", id = "chat")
```

The page supplies the full-window sizing and keeps `show_history = TRUE` on the mounted chat. Do not wrap `page_chat()` in another page container or pass `height`, `fill`, or `show_history`; those arguments are page-owned.

Navigation, sidebars, and artifacts

`pages_navbar` accepts a list of additional navbar items. Use `chat_nav_panel()` when a page needs page-chat-specific sidebar, toolbar, or content-width options. It also accepts `bslib::nav_panel()`, `bslib::nav_panel_hidden()`, `bslib::nav_menu()`, `bslib::nav_item()`, and `bslib::nav_spacer()`. Programmatic navigation uses standard bslib helpers against the derived "`<id>_page`" id: `bslib::nav_select()` to switch pages (including hidden panels and `nav_menu()` children), `bslib::nav_show()` and `bslib::nav_hide()` to reveal or hide nav controls. The active page is readable as `input$<id>_page` ("`__home__`" when the main chat page is active). Sidebar navigation is not yet implemented. Each panel can use the default sidebar, no page-specific sidebar, or its own `chat_sidebar()` or `bslib::sidebar()` configuration. The sidebar argument configures the home view. Use `bslib::toolbar()` to group controls in toolbar; it is a home-page-scoped segment rendered with the page navigation controls and follows them into the mobile app menu. A panel's `toolbar = NULL` omits that scoped segment; `chat_nav_panel(toolbar = bslib::toolbar(...))` supplies a page-specific replacement. Use `toolbar_global = bslib::toolbar(...)` for a persistent segment that remains mounted on every page after the active scoped toolbar. On narrow screens, navigation and toolbar controls move into the app menu above the active page's sidebar content without duplicating Shiny input or output IDs. By default, `toolbar_global` contains `bslib::input_dark_mode()`; use `NULL` to opt out.

Set drawer to a `chat_drawer()` configuration to provide initial content and layout options. Update the mounted drawer from the server with `chat_drawer_show()`, `chat_drawer_update()`, `chat_drawer_hide()`, and `chat_drawer_toggle()`. Artifact content is static UI passed through those server functions; use ordinary Shiny inputs and outputs inside that content when needed. You can try navigation and artifact-control examples, which do not require credentials, through `shiny::runExample("page-chat-navigation", package = "shinychat")` and `shiny::runExample("page-chat-draw", package = "shinychat")`.

`page_chat()` owns page composition and accepts one chat root. Do not pass unrelated top-level UI or a second chat root. Existing apps that need those layouts should continue using `chat_ui()` with `bslib::page_fillable()`, `bslib::page_sidebar()`, or another appropriate container.

Examples

```
library(shiny)
library(shinychat)

artifact_content <- function(label) {
  tags$div(
    tags$h3("Preview"),
    tags$p(label)
  )
}
```

```

    )
  }

  ui <- page_chat(
    "Assistant",
    messages = "Welcome! Ask a question to get started.",
    toolbar = bslib::toolbar(actionButton("show_preview", "Show preview")),
    toolbar_global = actionButton("help", "Help"),
    sidebar = chat_sidebar(
      tags$p("Home tools"),
      history = FALSE,
      open = "open"
    ),
    pages_navbar = list(
      chat_nav_panel(
        "About",
        tags$p("This is a secondary page."),
        value = "about",
      ),
      chat_nav_panel(
        "Settings",
        tags$p("Settings live here."),
        value = "settings",
        sidebar = chat_sidebar(
          tags$p("Settings menu"),
          width = 320,
          open = "closed"
        ),
        toolbar = bslib::toolbar(actionButton("save_settings", "Save settings"))
      )
    ),
    drawer = chat_drawer(
      artifact_content("Initial preview"),
      title = "Preview"
    )
  )

  server <- function(input, output, session) {
    observeEvent(input$chat_user_input, {
      chat_append("chat", paste0("You said: ", input$chat_user_input))
    })

    observeEvent(input$show_preview, {
      chat_drawer_show(
        "chat",
        content = artifact_content("Preview opened from the server"),
        title = "Preview"
      )
    })
  }

  shinyApp(ui, server)

```

page_chat_theme	Create a theme for page_chat()
-----------------	--------------------------------

Description

page_chat_theme() layers page-scoped surface, chat-radius, and density tokens and system typography over bslib's "shiny" preset. Supply a different preset to start from another bslib or Bootstrap preset, or pass a regular `bslib::bs_theme()` directly to `page_chat()` to omit the page-chat baseline.

Usage

```
page_chat_theme(..., preset = "shiny")
```

Arguments

...	Sass variables forwarded to <code>bslib::bs_theme()</code> . Values supplied here override the page-chat defaults.
preset	A bslib or Bootstrap preset name.

Value

A `bslib::bs_theme()` suitable for the theme argument of `page_chat()`.

tool_result_display	Customize how a tool result is displayed
---------------------	--

Description

tool_result_display() creates an object you can assign to the display item of the extra argument of an `ellmer::ContentToolResult` to customize how shinychat displays the tool result to the user, while keeping the underlying value sent to the model unchanged.

Usage

```
tool_result_display(
  title = NULL,
  icon = NULL,
  html = NULL,
  markdown = NULL,
  text = NULL,
  show_request = TRUE,
  open = FALSE,
  full_screen = FALSE,
  footer = NULL,
```

```

    label = NULL,
    value_preview = NULL,
    open_style = "minimal"
  )

```

Arguments

title	The title to use for the settled call and drill-down card. It replaces the definition-level title from <code>ellmer::tool_annotations()</code> in a single-call row. In a multi-call group, a distinct result title can identify the call in the expanded call list. Write the definition title in the present tense (for example, "Getting weather") and this result title in the past tense (for example, "Got weather").
icon	An icon to display with the settled call and drill-down card. Can be a character string or HTML content (e.g. from htmltools::tags).
html	Custom HTML content (to use in place of the default result content in the drill-down card).
markdown	Custom Markdown string (to use in place of the default result content in the drill-down card).
text	Custom plain text string (to use in place of the default result content in the drill-down card).
show_request	Whether to show the tool request inside the drill-down card.
open	Whether to open the drill-down card by default when the result settles.
full_screen	Whether or not to display a fullscreen toggle button on the drill-down card.
footer	Optional HTML content to display below the drill-down card body.
label	A short, per-call identifying value shown in the activity row (e.g. a filename or query). Distinguishes this call from other calls to the same tool. Without one, shinychat falls back to the call's own title (when it differs from the group's), then a short preview of the call's arguments, then the tool name.
value_preview	A terse, per-call preview of the tool result, shown in the activity row before its drill-down card is opened.
open_style	Whether the result uses the minimal drill-down style or a framed style when open.

Details

It preserves shinychat's compact activity row and drill-down card. Use it for result titles, per-call labels and previews, or rich card content. To replace the settled card with fully custom standalone UI, extend `contents_shinychat()` instead. See the [Tool Calling UI article](#) for a complete guide.

Value

An object of class `shinychat_tool_result_display`, for use as `extra = list(display = tool_result_display(...))` when creating an `ellmer::ContentToolResult`.

Examples

```
library(ellmer)

get_current_weather <- function(location) {
  ContentToolResult(
    value = "72 degrees and sunny",
    extra = list(
      display = tool_result_display(
        title = paste("Got weather for", location),
        label = location,
        value_preview = "72°F and sunny",
        markdown = "It's **72°F** and sunny."
      )
    )
  )
}
```

update_chat_user_input

Update the user input of a chat control

Description

Update the user input of a chat control

Usage

```
update_chat_user_input(
  id,
  ...,
  value = NULL,
  placeholder = NULL,
  submit = FALSE,
  focus = FALSE,
  attachments = NULL,
  attachment_mode = c("append", "set"),
  session = getDefaultReactiveDomain()
)
```

Arguments

id	The ID of the chat element
...	Currently unused, but reserved for future use.
value	The value to set the user input to. If NULL, the input will not be updated.
placeholder	The placeholder text for the user input
submit	Whether to automatically submit the text for the user. Requires value.

focus	Whether to move focus to the input element. Requires value or non-empty attachments.
attachments	A list of attachment objects created by <code>chat_attachment()</code> . When NULL (default), any existing staged attachments are left unchanged. Pass an empty list (<code>list()</code>) to clear staged attachments.
attachment_mode	How to combine attachments with any already-staged attachments. "append" (default) adds to the existing set; "set" replaces it. Use <code>attachment_mode = "set"</code> with <code>attachments = list()</code> to clear all staged attachments.
session	The Shiny session object

Examples

```
library(shiny)
library(bslib)
library(shinychat)

ui <- page_fillable(
  chat_ui("chat"),
  layout_columns(
    fill = FALSE,
    actionButton("update_placeholder", "Update placeholder"),
    actionButton("update_value", "Update user input")
  )
)

server <- function(input, output, session) {
  observeEvent(input$update_placeholder, {
    update_chat_user_input("chat", placeholder = "New placeholder text")
  })

  observeEvent(input$update_value, {
    update_chat_user_input("chat", value = "New user input", focus = TRUE)
  })

  observeEvent(input$chat_user_input, {
    response <- paste0("You said: ", input$chat_user_input)
    chat_append("chat", response)
  })
}

shinyApp(ui, server)
```

Index

- * **chat drawers**
 - chat_drawer, 15
 - chat_drawer_hide, 16
 - chat_drawer_show, 17
 - chat_drawer_toggle, 17
 - chat_drawer_update, 18
- * **tool display**
 - tool_result_display, 51
- an S7 generic, 37
- bsicons::bs_icon(), 33
- bslib::bs_theme(), 48, 51
- bslib::input_dark_mode(), 48, 49
- bslib::nav_hide(), 48, 49
- bslib::nav_item(), 49
- bslib::nav_menu(), 49
- bslib::nav_panel(), 22, 49
- bslib::nav_panel_hidden(), 48, 49
- bslib::nav_select(), 48, 49
- bslib::nav_show(), 48, 49
- bslib::nav_spacer(), 49
- bslib::navbar_options(), 48
- bslib::page_fillable(), 49
- bslib::page_navbar(), 22
- bslib::page_sidebar(), 49
- bslib::sidebar(), 22, 27, 48, 49
- bslib::toolbar(), 23, 29, 48, 49
- chat_app, 2
- chat_app(), 47
- chat_append, 7
- chat_append(), 4, 11, 28, 30, 44
- chat_append_message, 11
- chat_attachment, 13
- chat_attachment(), 54
- chat_clear, 14
- chat_clear(), 32
- chat_drawer, 15
- chat_drawer(), 16–18, 29, 49
- chat_drawer_hide, 16
- chat_drawer_hide(), 16–18, 49
- chat_drawer_show, 17
- chat_drawer_show(), 16, 18, 49
- chat_drawer_toggle, 17
- chat_drawer_toggle(), 16–18, 49
- chat_drawer_update, 18
- chat_drawer_update(), 16–18, 49
- chat_enable_history, 19
- chat_enable_history(), 23, 39, 44
- chat_get_greeting, 20
- chat_greeting, 20
- chat_greeting(), 3, 4, 6, 25, 29, 32
- chat_nav_panel, 22
- chat_nav_panel(), 27, 28, 48, 49
- chat_restore, 23
- chat_server (chat_app), 2
- chat_server(), 3, 5, 14, 30–32, 37, 47
- chat_set_greeting, 25
- chat_set_greeting(), 20, 32
- chat_sidebar, 27
- chat_sidebar(), 22, 48, 49
- chat_ui, 28
- chat_ui(), 2, 5, 7, 14–18, 20, 25, 33, 47–49
- chat_ui_history, 35
- contents_shinychat, 37
- contents_shinychat(), 52
- ContentSlashCommand, 5, 36
- ConversationStore, 39, 41, 43
- coro::async_generator(), 44
- coro::generator(), 44
- ellmer::Chat, 2, 19
- ellmer::chat_openai(), 3
- ellmer::Content, 30, 37
- ellmer::contents_record(), 36
- ellmer::contents_replay(), 36
- ellmer::ContentText, 36
- ellmer::ContentToolResult, 37, 51, 52
- ellmer::tool(), 38

FileConversationStore, 41

history_options, 43
history_options(), 3, 19
htmltools::HTML(), 3, 8, 21, 25, 30, 31
htmltools::img(), 8, 12
htmltools::tagList(), 3, 8, 17, 18, 21, 31, 33
htmltools::tags, 52
htmltools::tags(), 30

markdown_stream, 44
markdown_stream(), 46

output_markdown_stream, 45
output_markdown_stream(), 44

page_chat, 47
page_chat(), 3, 5, 15–18, 22, 23, 27, 28, 51
page_chat_theme, 51
page_chat_theme(), 48
promises::promise(), 44

rlang::inject(), 30

S7::super(), 38
shiny::ExtendedTask, 4
shiny::observeEvent(), 25, 32
shiny::shinyApp(), 3–5

tool_result_display, 51
tool_result_display(), 37, 39

update_chat_user_input, 53
update_chat_user_input(), 4, 13