

Package: shinychat (via r-universe)

June 5, 2026

Title Chat UI Component for 'shiny'

Version 0.4.0.9000

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, cli, coro, ellmer (>= 0.4.1), fastmap, htmltools, jsonlite, lifecycle, promises (>= 1.3.2), rlang (>= 1.1.0), S7, shiny (>= 1.10.0)

Suggests covr, knitr, later, R6, testthat (>= 3.0.0), withr

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

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

Config/pak/sysreqs cmake make libuv1-dev libssl-dev zlib1g-dev

Repository <https://posit-dev.r-universe.dev>

Date/Publication 2026-06-05 16:21:46 UTC

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

RemoteRef HEAD

RemoteSha dc2a33a78e44f3960978fcd996ee9ccf6d892b7a

RemoteSubdir pkg-r

Contents

chat_app	2
chat_append	5
chat_append_message	7
chat_clear	9
chat_greeting	11
chat_restore	12
chat_set_greeting	14
chat_ui	17
contents_shinychat	20
markdown_stream	22
output_markdown_stream	23
update_chat_user_input	24

Index	27
--------------	-----------

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 the Shiny module chat functions – `chat_mod_ui()` and `chat_mod_server()` – and be sure to create a new chat client for each user session.

Usage

```
chat_app(client, ..., bookmark_store = "url")

chat_mod_ui(id, ..., client = deprecated(), messages = NULL)

chat_mod_server(
  id,
  client,
  greeting = NULL,
  bookmark_on_input = TRUE,
  bookmark_on_response = TRUE
)
```

Arguments

<code>client</code>	A chat object created by ellmer , e.g. <code>ellmer::chat_openai()</code> and friends. This argument is deprecated in <code>chat_mod_ui()</code> because the client state is now managed by <code>chat_mod_server()</code> .
---------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

...	In <code>chat_app()</code> , additional arguments are passed to <code>shiny::shinyApp()</code> . In <code>chat_mod_ui()</code> , additional arguments are passed to <code>chat_ui()</code> .
<code>bookmark_store</code>	The bookmarking store to use for the app. Passed to <code>enable_bookmarking</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".
<code>id</code>	The chat module ID.
<code>messages</code>	Initial messages shown in the chat, used only when <code>client</code> (in <code>chat_mod_ui()</code>) doesn't already contain turns. Passed to <code>messages</code> in <code>chat_ui()</code> .
<code>greeting</code>	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.
<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.

Value

- `chat_app()` returns a `shiny::shinyApp()` object.
- `chat_mod_ui()` returns the UI for a shinychat module.
- `chat_mod_server()` includes the shinychat module server logic, and returns an environment containing:
 - `last_input`: A reactive value containing the last user input.
 - `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 history and the chat UI. `clear()` takes an optional 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.
 - `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.
- `client`: The current chat client object (an active binding that always reflects the latest client, even after `set_client()` is called).
- `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.

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_mod_ui()`: A simple chat app module UI.
- `chat_mod_server()`: A simple chat app module server.

Greeting

When greeting is a **function**, the module calls it 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 module detects **named arguments** in the greeting function to decide what to pass. Currently the only recognized argument is `client`. `function(client)` (recommended). The module clones the `client` passed to `chat_mod_server()`, wipes its turn history, and passes the fresh clone as `client`. This avoids manually creating and configuring a separate client:

```
chat_mod_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_mod_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_mod_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_mod_ui(
        "claude",
        messages = list(
          "Hi! Use this chat interface to chat with Anthropic's `claude-3-5-sonnet`."
        )
      )
    ),
    card(
      card_header("Chat with ChatGPT"),
      chat_mod_ui(
        "openai",
        messages = list(
          "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_mod_server("claude", claude)
  chat_mod_server("openai", openai)
}

shinyApp(ui, server)

## End(Not run)

```

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: <ul style="list-style-type: none"> • A string, which is interpreted as markdown and rendered to HTML on the client. <ul style="list-style-type: none"> – To prevent interpreting as markdown, mark the string as <code>htmltools::HTML()</code>. • A UI element. <ul style="list-style-type: none"> – This includes <code>htmltools::tagList()</code>, which take UI elements (including strings) as children. In this case, strings are still interpreted as markdown as long as they're not inside HTML.
<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.
<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.

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)
```

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.
<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_clear

Clear all messages from a chat control

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()`).

Usage

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

Arguments

id	The ID of the chat element
greeting	If TRUE, also clears the greeting. When the greeting is cleared, greeting_requested will fire again (if the chat is visible), allowing the server to generate a new greeting.
session	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_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, dismissible = TRUE)
```

Arguments

content	The greeting content. Can be: <ul style="list-style-type: none"> • A string, interpreted as markdown. • An <code>htmltools::HTML()</code> object, rendered as raw HTML. • An htmltools tag or <code>htmltools::tagList()</code>, including Shiny inputs/outputs. • A generator or promise (only valid when used with <code>chat_set_greeting()</code>).
dismissible	Whether the greeting is automatically dismissed when the user sends a message. Defaults to TRUE.

Value

An S3 object of class "chat_greeting".

Patterns

Non-dismissible greeting (stays visible after the user sends a message):

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

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_restore

Add Shiny bookmarking for shinychat

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: Only the client's chat state is saved/restored in the bookmark. 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), then 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()`.

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 *Set the greeting for a chat UI*

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:

- `NULL` to clear the current greeting.
 - A plain string or `htmltools::HTML()` object (auto-wrapped with default options).
 - A `chat_greeting()` object.
- session 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_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,
  ...,
  messages = NULL,
  greeting = NULL,
  placeholder = "Enter a message...",
  width = "min(680px, 100%)",
  height = "auto",
  fill = TRUE,
  icon_assistant = NULL,
  enable_cancel = FALSE,
  footer = NULL
)

```

Arguments

<code>id</code>	The ID of the chat element
<code>...</code>	Extra HTML attributes to include on the chat element
<code>messages</code>	A list of messages to prepopulate the chat with. Each message can be one of the following: <ul style="list-style-type: none"> A string, which is interpreted as markdown and rendered to HTML on the client. <ul style="list-style-type: none"> To prevent interpreting as markdown, mark the string as <code>htmltools::HTML()</code>. A UI element.

- This includes `htmltools::tagList()`, which take UI elements (including strings) as children. In this case, strings are still interpreted as markdown as long as they're not inside HTML.
- A named list of content and role. The content can contain content as described above, and the role can be "assistant" or "user".

greeting	An optional greeting to display when the chat first loads. Can be a <code>chat_greeting()</code> 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>
placeholder	The placeholder text for the chat's user input field
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> . If None, a default robot icon is used.
enable_cancel	If TRUE, show a stop button during streaming that allows the user to cancel the in-progress response. When using <code>chat_mod_server()</code> , cancellation is wired up automatically. For manual usage with <code>chat_ui()</code> , observe <code>input\$<id>_cancel</code> to handle cancellation (e.g., by calling <code>ctrl\$cancel()</code> on an <code>ellmer stream_controller()</code>). Defaults to FALSE.
footer	Optional HTML content to display below the chat input. 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.

Value

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

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 `dismissible = FALSE`).

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.

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.

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)
    chat_append("chat", stream)
  })
}

shinyApp(ui, server)
```

contents_shinychat *Format ellmer content for shinychat*

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](#).

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, we can extend `contents_shinychat()` to render our custom content class for display in the chat interface. The basic process is to define a `contents_shinychat()` external generic and then 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
}
```

You can use this pattern to completely customize how the content is displayed inside shinychat by returning HTML objects directly from this method.

You can also use this pattern to build upon the default shinychat display for tool requests and results. By using `S7::super()`, you can create the object shinychat uses for tool results (or tool requests), and then modify it to suit your needs.

```

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("Weather Forecast for", content@location_name)

  res
}

```

Note that you do **not** need to create a new class or extend `contents_shinychat()` to customize the tool display. Rather, you can use the strategies discussed in the [Tool Calling UI article](#) to customize the tool request and result display by providing a display list in the extra argument of the tool result.

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

<code>id</code>	The ID of the markdown stream to stream content to.
<code>content_stream</code>	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.
<code>operation</code>	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.
<code>session</code>	The Shiny session object.

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

id	A unique identifier for this markdown stream.
...	Extra HTML attributes to include on the chat element
content	A string of content to display before any streaming occurs. When content_type is Markdown or HTML, it may also be UI element(s) such as input and output bindings.
content_type	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.
auto_scroll	Whether to automatically scroll to the bottom of a scrollable container when new content is added. Default is True.
width	The width of the UI element.
height	The height of the UI element.

Value

A shiny tag object.

See Also

[markdown_stream\(\)](#)

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,
  session = getDefaultReactiveDomain()
)
```

Arguments

<code>id</code>	The ID of the chat element
<code>...</code>	Currently unused, but reserved for future use.
<code>value</code>	The value to set the user input to. If <code>NULL</code> , the input will not be updated.
<code>placeholder</code>	The placeholder text for the user input
<code>submit</code>	Whether to automatically submit the text for the user. Requires <code>value</code> .
<code>focus</code>	Whether to move focus to the input element. Requires <code>value</code> .
<code>session</code>	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)
  })
}
```

26

update_chat_user_input

}

shinyApp(ui, server)

Index

an S7 generic, [20](#)

chat_app, [2](#)
chat_append, [5](#)
chat_append(), [3](#), [8](#), [17](#)
chat_append_message, [7](#)
chat_clear, [9](#)
chat_clear(), [19](#)
chat_greeting, [11](#)
chat_greeting(), [3](#), [4](#), [15](#), [18](#)
chat_mod_server (chat_app), [2](#)
chat_mod_server(), [18](#)
chat_mod_ui (chat_app), [2](#)
chat_restore, [12](#)
chat_set_greeting, [14](#)
chat_set_greeting(), [11](#), [18](#)
chat_ui, [17](#)
chat_ui(), [3](#), [6](#), [9](#), [11](#), [14](#)
contents_shinychat, [20](#)
coro::async_generator(), [22](#)
coro::generator(), [22](#)

ellmer::Chat, [2](#)
ellmer::chat_openai(), [2](#)
ellmer::Content, [20](#)
ellmer::ContentToolResult, [20](#)
ellmer::tool(), [21](#)

htmltools::HTML(), [3](#), [6](#), [11](#), [15](#), [17](#), [18](#)
htmltools::img(), [6](#), [8](#)
htmltools::tagList(), [3](#), [6](#), [11](#), [18](#)
htmltools::tags(), [18](#)

markdown_stream, [22](#)
markdown_stream(), [23](#), [24](#)

output_markdown_stream, [23](#)
output_markdown_stream(), [22](#)

promises::promise(), [22](#)

S7::super(), [21](#)
shiny::ExtendedTask, [3](#)
shiny::observeEvent(), [14](#), [19](#)
shiny::shinyApp(), [3](#)

update_chat_user_input, [24](#)
update_chat_user_input(), [3](#)