Package 'shinychat'

May 16, 2025

Title Chat UI Component for 'shiny'

Version 0.2.0

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://github.com/posit-dev/shinychat, https://posit-dev.github.io/shinychat/

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

Imports bslib, coro, ellmer, fastmap, htmltools, jsonlite, promises (>= 1.3.2), rlang, shiny (>= 1.10.0)

Suggests later, testthat (>= 3.0.0)

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2025-05-16 08:50:02 UTC

Contents

`chat_app` *Open a live chat application in the browser*

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.

Usage

```
chat_app(client, ...)

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

chat_mod_server(id, client)
```

Arguments

`client`	A chat object created by **ellmer**, e.g. `ellmer::chat_openai()` and friends.
`...`	In `chat_app()`, additional arguments are passed to `shiny::shinyApp()`. In `chat_mod_ui()`, additional arguments are passed to `chat_ui()`.
`id`	The chat module ID.
`messages`	Initial messages shown in the chat, used when `client` is not provided or when the chat `client` doesn't already contain turns. Passed to `messages` in `chat_ui()`.

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 and returns the last turn upon successful chat completion.

Functions

- `chat_app()`: A simple Shiny app for live chatting.
- `chat_mod_ui()`: A simple chat app module UI.
- `chat_mod_server()`: A simple chat app module server.

Examples

```
## Not run:
# Interactive in the console ----
client <- ellmer::chat_claude()
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_claude(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)
```

---

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 model'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"),
  session = getDefaultReactiveDomain()
)
```

Arguments

id	The ID of the chat element

response	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 htmltools::HTML().
	• 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.

role	The role of the message (either "assistant" or "user"). Defaults to "assistant".
session	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 will see that the observer failed, and end the user session. If you prefer to handle the error gracefully, use `promises::catch()` on the promise returned by `chat_append`.

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"),
  session = getDefaultReactiveDomain()
)
```

Arguments

`id`	The ID of the chat element
`msg`	The message to append. Should be a named list with `role` and `content` fields. The `role` field should be either "user" or "assistant". The `content` field should be a string containing the message content, in Markdown format.
`chunk`	Whether `msg` is just a chunk of a message, and if so, what type. If `FALSE`, then `msg` is a complete message. If `"start"`, then `msg` is the first chunk of a multi-chunk message. If `"end"`, then `msg` is the last chunk of a multi-chunk message. If `TRUE`, then `msg` is an intermediate chunk of a multi-chunk message. Default is `FALSE`.
`operation`	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 `chunk` is `FALSE`.
`session`	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

Clear all messages from a chat control

Usage

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

Arguments

`id`	The ID of the chat element
`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)
```

---

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,
  placeholder = "Enter a message...",
  width = "min(680px, 100%)",
  height = "auto",
  fill = TRUE
)
```

Arguments

`id`	The ID of the chat element
`...`	Extra HTML attributes to include on the chat element

`messages`	A list of messages to prepopulate the chat with. 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 `htmltools::HTML()`.
	• 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".

`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

Value

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

Examples

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

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

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

```
shinyApp(ui, server)
```

`markdown_stream` *Stream markdown content*

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., `coro::generator()` or `coro::async_generator()`), a string promise (e.g., `promises::promise()`), or a string promise generator.
`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.

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. * "semi-markdown": for rendering markdown, but with HTML tags escaped.
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()

Index