1. Creación de agentes utilizando Python.

A screenshot of a computer

AI-generated content may be incorrect.

from fastapi.middleware.cors import CORSMiddleware

from agno.agent import Agent

from agno.models.groq import Groq

from agno.playground import Playground

from agno.storage.sqlite import SqliteStorage

from agno.tools.duckduckgo import DuckDuckGoTools

from agno.tools.yfinance import YFinanceTools

agent\_storage: str = "tmp/agents.db"

web\_agent = Agent(

*name*="Web Agent",

*model*=Groq(*id*="llama-3.3-70b-versatile"),

*tools*=[DuckDuckGoTools()],

*instructions*=["Always include sources"],

*storage*=SqliteStorage(*table\_name*="web\_agent", *db\_file*=agent\_storage),

*add\_datetime\_to\_instructions*=True,

*add\_history\_to\_messages*=True,

*num\_history\_responses*=5,

*markdown*=True,

)

finance\_agent = Agent(

*name*="Finance Agent",

*model*=Groq(*id*="llama-3.3-70b-versatile"),

*tools*=[YFinanceTools(

*stock\_price*=True,

*analyst\_recommendations*=True,

*company\_info*=True,

*company\_news*=True

    )],

*instructions*=["Always use tables to display data"],

*storage*=SqliteStorage(*table\_name*="finance\_agent", *db\_file*=agent\_storage),

*add\_datetime\_to\_instructions*=True,

*add\_history\_to\_messages*=True,

*num\_history\_responses*=5,

*markdown*=True,

)

playground\_app = Playground(*agents*=[web\_agent, finance\_agent])

app = playground\_app.get\_app()

*# ✅ Permitir cualquier origen CORS*

app.add\_middleware(

    CORSMiddleware,

*allow\_origins*=["\*"],  *# ⚠️ En producción, es recomendable restringir esto*

*allow\_credentials*=True,

*allow\_methods*=["\*"],

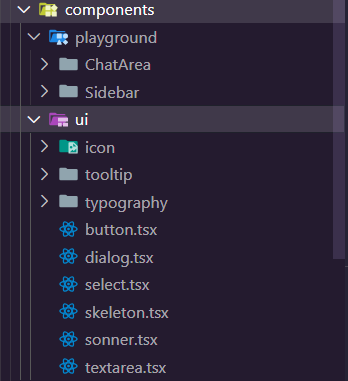
*allow\_headers*=["\*"],

)

if \_\_name\_\_ == "\_\_main\_\_":

    playground\_app.serve("main:app", *reload*=True)

1. Enfoque en la creación de componentes.



'use client'

import ChatInput from './ChatInput'

import MessageArea from './MessageArea'

const ChatArea = () => {

  return (

    <main className="relative m-1.5 flex flex-grow flex-col rounded-xl ">

      <MessageArea />

      <div className="sticky bottom-0 ml-9 px-4 pb-2">

        <ChatInput />

      </div>

    </main>

  )

}

export default ChatArea

"use client"

import type React from "react"

import { motion, AnimatePresence } from "framer-motion"

import { useStickToBottomContext } from "use-stick-to-bottom"

import { Button } from "@/components/ui/button"

import Icon from "@/components/ui/icon"

const ScrollToBottom: React.FC = () => {

  const { isAtBottom, scrollToBottom } = useStickToBottomContext()

  return (

    <AnimatePresence>

      {!isAtBottom && (

        <motion.div

          initial={{ opacity: 0, y: 20, scale: 0.8 }}

          animate={{ opacity: 1, y: 0, scale: 1 }}

          exit={{ opacity: 0, y: 20, scale: 0.8 }}

          transition={{

            duration: 0.3,

            ease: "easeOut",

            type: "spring",

            stiffness: 400,

            damping: 25,

          }}

          className="absolute bottom-6 left-1/2 z-10 -translate-x-1/2"

        >

          <motion.div

            whileHover={{ scale: 1.05, y: -2 }}

            whileTap={{ scale: 0.95 }}

            transition={{ type: "spring", stiffness: 400, damping: 17 }}

          >

            <Button

              onClick={() => scrollToBottom()}

              type="button"

              size="icon"

              className="group relative h-12 w-12 overflow-hidden rounded-full border border-border/50 bg-background/95 backdrop-blur-sm text-primary shadow-lg transition-all duration-300 hover:border-primary/30 hover:bg-background hover:shadow-xl"

            >

              {*/\* Background gradient on hover \*/*}

              <div className="absolute inset-0 bg-gradient-to-br from-primary/10 to-primary/5 opacity-0 transition-opacity duration-300 group-hover:opacity-100" />

              {*/\* Icon with bounce animation \*/*}

              <motion.div

                animate={{ y: [0, -2, 0] }}

                transition={{

                  duration: 1.5,

                  repeat: Number.POSITIVE\_INFINITY,

                  ease: "easeInOut",

                }}

                className="relative z-10"

              >

                <Icon type="arrow-down" size="sm" className="text-primary" />

              </motion.div>

              {*/\* Ripple effect \*/*}

              <div className="absolute inset-0 rounded-full bg-primary/20 opacity-0 transition-opacity duration-300 group-active:opacity-100" />

            </Button>

          </motion.div>

          {*/\* Subtle tooltip \*/*}

          <motion.div

            initial={{ opacity: 0, y: 10 }}

            animate={{ opacity: 1, y: 0 }}

            transition={{ delay: 0.5, duration: 0.3 }}

            className="absolute -top-12 left-1/2 -translate-x-1/2 whitespace-nowrap rounded-lg bg-popover/95 backdrop-blur-sm px-3 py-1.5 text-xs font-medium text-popover-foreground shadow-md"

          >

            Scroll to bottom

            <div className="absolute -bottom-1 left-1/2 h-2 w-2 -translate-x-1/2 rotate-45 bg-popover/95" />

          </motion.div>

        </motion.div>

      )}

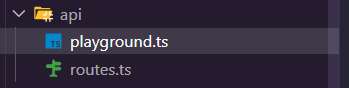
    </AnimatePresence>

  )

}

export default ScrollToBottom

1. Uso de peticiones **fetch**.



import { toast } from 'sonner'

import { APIRoutes } from './routes'

import { Agent, ComboboxAgent, SessionEntry } from '@/types/playground'

export const getPlaygroundAgentsAPI = async (

*endpoint*: string

): Promise<ComboboxAgent[]> => {

  const url = APIRoutes.GetPlaygroundAgents(*endpoint*)

  try {

    const response = await fetch(url, { method: 'GET' })

    if (!response.ok) {

      toast.error(`Failed to fetch playground agents: ${response.statusText}`)

      return []

    }

    const data = await response.json()

*// Transform the API response into the expected shape.*

    const agents: ComboboxAgent[] = data.map((*item*: Agent) => ({

      value: *item*.agent\_id || '',

      label: *item*.name || '',

      model: *item*.model || '',

      storage: *item*.storage || false

    }))

    return agents

  } catch {

    toast.error('Error fetching playground agents')

    return []

  }

}

export const getPlaygroundStatusAPI = async (*base*: string): Promise<number> => {

  const response = await fetch(APIRoutes.PlaygroundStatus(*base*), {

    method: 'GET'

  })

  return response.status

}

export const getAllPlaygroundSessionsAPI = async (

*base*: string,

*agentId*: string

): Promise<SessionEntry[]> => {

  try {

    const response = await fetch(

      APIRoutes.GetPlaygroundSessions(*base*, *agentId*),

      {

        method: 'GET'

      }

    )

    if (!response.ok) {

      if (response.status === 404) {

*// Return empty array when storage is not enabled*

        return []

      }

      throw new Error(`Failed to fetch sessions: ${response.statusText}`)

    }

    return response.json()

  } catch {

    return []

  }

}

export const getPlaygroundSessionAPI = async (

*base*: string,

*agentId*: string,

*sessionId*: string

) => {

  const response = await fetch(

    APIRoutes.GetPlaygroundSession(base, agentId, sessionId),

    {

      method: 'GET'

    }

  )

  return response.json()

}

export const deletePlaygroundSessionAPI = async (

*base*: string,

*agentId*: string,

*sessionId*: string

) => {

  const response = await fetch(

    APIRoutes.DeletePlaygroundSession(base, agentId, sessionId),

    {

      method: 'DELETE'

    }

  )

  return response

}

1. Implementación de **streaming** mediante SSE (Server-Sent Events).



import { useCallback } from 'react'

import { type RunResponse } from '@/types/playground'

*/\*\**

*\* Processes a single JSON chunk by passing it to the provided callback.*

*\* @param chunk - A parsed JSON object of type RunResponse.*

*\* @param onChunk - Callback to handle the chunk.*

*\*/*

function processChunk(

*chunk*: RunResponse,

  onChunk: (*chunk*: RunResponse) => void

) {

  onChunk(*chunk*)

}

*/\*\**

*\* Parses a string buffer to extract complete JSON objects.*

*\**

*\* This function discards any extraneous data before the first '{', then*

*\* repeatedly finds and processes complete JSON objects.*

*\**

*\* @param text - The accumulated string buffer.*

*\* @param onChunk - Callback to process each parsed JSON object.*

*\* @returns Remaining string that did not form a complete JSON object.*

*\*/*

*/\*\**

*\* Extracts complete JSON objects from a buffer string \*\*incrementally\*\*.*

*\* - It allows partial JSON objects to accumulate across chunks.*

*\* - It ensures real-time streaming updates.*

*\*/*

function parseBuffer(

*buffer*: string,

  onChunk: (*chunk*: RunResponse) => void

): string {

  let jsonStartIndex = *buffer*.indexOf('{')

  let jsonEndIndex = -1

  while (jsonStartIndex !== -1) {

    let braceCount = 0

    let inString = false

*// Iterate through the buffer to find the end of the JSON object*

    for (let i = jsonStartIndex; i < *buffer*.length; i++) {

      const char = *buffer*[i]

*// Check if the character is a double quote and the previous character is not a backslash*

*// This is to handle escaped quotes in JSON strings*

      if (char === '"' && *buffer*[i - 1] !== '\\') {

        inString = !inString

      }

*// If the character is not inside a string, count the braces*

      if (!inString) {

        if (char === '{') braceCount++

        if (char === '}') braceCount--

      }

*// If the brace count is 0, we have found the end of the JSON object*

      if (braceCount === 0) {

        jsonEndIndex = i

        break

      }

    }

*// If we found a complete JSON object, process it*

    if (jsonEndIndex !== -1) {

      const jsonString = *buffer*.slice(jsonStartIndex, jsonEndIndex + 1)

      try {

        const parsed = JSON.parse(jsonString) as RunResponse

        processChunk(parsed, onChunk)

      } catch {

*// Skip invalid JSON, continue accumulating*

        break

      }

*buffer* = *buffer*.slice(jsonEndIndex + 1).trim()

      jsonStartIndex = *buffer*.indexOf('{')

      jsonEndIndex = -1

    } else {

*// No complete JSON found, wait for the next chunk*

      break

    }

  }

  return buffer

}

*/\*\**

*\* Custom React hook to handle streaming API responses as JSON objects.*

*\**

*\* This hook:*

*\* - Accumulates partial JSON data from streaming responses.*

*\* - Extracts complete JSON objects and processes them via onChunk.*

*\* - Handles errors via onError and signals completion with onComplete.*

*\**

*\* @returns An object containing the streamResponse function.*

*\*/*

export default function useAIResponseStream() {

  const streamResponse = useCallback(

    async (*options*: {

      apiUrl: string

      headers?: Record<string, string>

      requestBody: FormData | Record<string, unknown>

      onChunk: (*chunk*: RunResponse) => void

      onError: (*error*: Error) => void

      onComplete: () => void

    }): Promise<void> => {

      const {

        apiUrl,

        headers = {},

        requestBody,

        onChunk,

        onError,

        onComplete

      } = options

*// Buffer to accumulate partial JSON data.*

      let buffer = ''

      try {

        const response = await fetch(apiUrl, {

          method: 'POST',

          headers: {

*// Set content-type only for non-FormData requests.*

            ...(!(requestBody instanceof FormData) && {

              'Content-Type': 'application/json'

            }),

            ...headers

          },

          body:

            requestBody instanceof FormData

              ? requestBody

              : JSON.stringify(requestBody)

        })

        if (!response.ok) {

          const errorData = await response.json()

          throw errorData

        }

        if (!response.body) {

          throw new Error('No response body')

        }

        const reader = response.body.getReader()

        const decoder = new TextDecoder()

*// Recursively process the stream.*

        const processStream = async (): Promise<void> => {

          const { done, value } = await reader.read()

          if (done) {

*// Process any final data in the buffer.*

            buffer = parseBuffer(buffer, onChunk)

            onComplete()

            return

          }

*// Decode, sanitize, and accumulate the chunk*

          buffer += decoder.decode(value, { stream: true })

*// Parse any complete JSON objects available in the buffer.*

          buffer = parseBuffer(buffer, onChunk)

          await processStream()

        }

        await processStream()

      } catch (error) {

        if (typeof error === 'object' && error !== null && 'detail' in error) {

          onError(new Error(String(error.detail)))

        } else {

          onError(new Error(String(error)))

        }

      }

    },

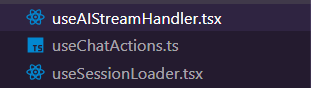
    []

  )

  return { streamResponse }

}

1. Uso de **hooks**.



import { useCallback } from 'react'

import { APIRoutes } from '@/api/routes'

import useChatActions from '@/hooks/useChatActions'

import { usePlaygroundStore } from '../store'

import {

  RunEvent,

  RunResponseContent,

  type RunResponse

} from '@/types/playground'

import { constructEndpointUrl } from '@/lib/constructEndpointUrl'

import useAIResponseStream from './useAIResponseStream'

import { ToolCall } from '@/types/playground'

import { useQueryState } from 'nuqs'

import { getJsonMarkdown } from '@/lib/utils'

*/\*\**

*\* useAIChatStreamHandler is responsible for making API calls and handling the stream response.*

*\* For now, it only streams message content and updates the messages state.*

*\*/*

const useAIChatStreamHandler = () => {

  const setMessages = usePlaygroundStore((*state*) => state.setMessages)

  const { addMessage, focusChatInput } = useChatActions()

  const [agentId] = useQueryState('agent')

  const [sessionId, setSessionId] = useQueryState('session')

  const selectedEndpoint = usePlaygroundStore((*state*) => state.selectedEndpoint)

  const setStreamingErrorMessage = usePlaygroundStore(

    (*state*) => state.setStreamingErrorMessage

  )

  const setIsStreaming = usePlaygroundStore((*state*) => state.setIsStreaming)

  const setSessionsData = usePlaygroundStore((*state*) => state.setSessionsData)

  const hasStorage = usePlaygroundStore((*state*) => state.hasStorage)

  const { streamResponse } = useAIResponseStream()

  const updateMessagesWithErrorState = useCallback(() => {

    setMessages((*prevMessages*) => {

      const newMessages = [...prevMessages]

      const lastMessage = newMessages[newMessages.length - 1]

      if (lastMessage && lastMessage.role === 'agent') {

        lastMessage.streamingError = true

      }

      return newMessages

    })

  }, [setMessages])

*/\*\**

*\* Processes a new tool call and adds it to the message*

*\* @param toolCall - The tool call to add*

*\* @param prevToolCalls - The previous tool calls array*

*\* @returns Updated tool calls array*

*\*/*

  const processToolCall = useCallback(

    (*toolCall*: ToolCall, *prevToolCalls*: ToolCall[] = []) => {

      const toolCallId =

        toolCall.tool\_call\_id || `${toolCall.tool\_name}-${toolCall.created\_at}`

      const existingToolCallIndex = prevToolCalls.findIndex(

        (*tc*) =>

          (tc.tool\_call\_id && tc.tool\_call\_id === toolCall.tool\_call\_id) ||

          (!tc.tool\_call\_id &&

            toolCall.tool\_name &&

            toolCall.created\_at &&

            `${tc.tool\_name}-${tc.created\_at}` === toolCallId)

      )

      if (existingToolCallIndex >= 0) {

        const updatedToolCalls = [...prevToolCalls]

        updatedToolCalls[existingToolCallIndex] = {

          ...updatedToolCalls[existingToolCallIndex],

          ...toolCall

        }

        return updatedToolCalls

      } else {

        return [...prevToolCalls, toolCall]

      }

    },

    []

  )

*/\*\**

*\* Processes tool calls from a chunk, handling both single tool object and tools array formats*

*\* @param chunk - The chunk containing tool call data*

*\* @param existingToolCalls - The existing tool calls array*

*\* @returns Updated tool calls array*

*\*/*

  const processChunkToolCalls = useCallback(

    (

      chunk: RunResponseContent | RunResponse,

      existingToolCalls: ToolCall[] = []

    ) => {

      let updatedToolCalls = [...existingToolCalls]

*// Handle new single tool object format*

      if (chunk.tool) {

        updatedToolCalls = processToolCall(chunk.tool, updatedToolCalls)

      }

*// Handle legacy tools array format*

      if (chunk.tools && chunk.tools.length > 0) {

        for (const toolCall of chunk.tools) {

          updatedToolCalls = processToolCall(toolCall, updatedToolCalls)

        }

      }

      return updatedToolCalls

    },

    [processToolCall]

  )

  const handleStreamResponse = useCallback(

    async (*input*: string | FormData) => {

      setIsStreaming(true)

      const formData = input instanceof FormData ? input : new FormData()

      if (typeof input === 'string') {

        formData.append('message', input)

      }

      setMessages((*prevMessages*) => {

        if (prevMessages.length >= 2) {

          const lastMessage = prevMessages[prevMessages.length - 1]

          const secondLastMessage = prevMessages[prevMessages.length - 2]

          if (

            lastMessage.role === 'agent' &&

            lastMessage.streamingError &&

            secondLastMessage.role === 'user'

          ) {

            return prevMessages.slice(0, -2)

          }

        }

        return prevMessages

      })

      addMessage({

        role: 'user',

        content: formData.get('message') as string,

        created\_at: Math.floor(Date.now() / 1000)

      })

      addMessage({

        role: 'agent',

        content: '',

        tool\_calls: [],

        streamingError: false,

        created\_at: Math.floor(Date.now() / 1000) + 1

      })

      let lastContent = ''

      let newSessionId = sessionId

      try {

        const endpointUrl = constructEndpointUrl(selectedEndpoint)

        if (!agentId) return

        const playgroundRunUrl = APIRoutes.AgentRun(endpointUrl).replace(

          '{agent\_id}',

          agentId

        )

        formData.append('stream', 'true')

        formData.append('session\_id', sessionId ?? '')

        await streamResponse({

          apiUrl: playgroundRunUrl,

          requestBody: formData,

          onChunk: (*chunk*: RunResponse) => {

            if (

              chunk.event === RunEvent.RunStarted ||

              chunk.event === RunEvent.ReasoningStarted

            ) {

              newSessionId = chunk.session\_id as string

              setSessionId(chunk.session\_id as string)

              if (

                hasStorage &&

                (!sessionId || sessionId !== chunk.session\_id) &&

                chunk.session\_id

              ) {

                const sessionData = {

                  session\_id: chunk.session\_id as string,

                  title: formData.get('message') as string,

                  created\_at: chunk.created\_at

                }

                setSessionsData((*prevSessionsData*) => {

                  const sessionExists = prevSessionsData?.some(

                    (*session*) => session.session\_id === chunk.session\_id

                  )

                  if (sessionExists) {

                    return prevSessionsData

                  }

                  return [sessionData, ...(prevSessionsData ?? [])]

                })

              }

            } else if (chunk.event === RunEvent.ToolCallStarted) {

              setMessages((*prevMessages*) => {

                const newMessages = [...prevMessages]

                const lastMessage = newMessages[newMessages.length - 1]

                if (lastMessage && lastMessage.role === 'agent') {

                  lastMessage.tool\_calls = processChunkToolCalls(

                    chunk,

                    lastMessage.tool\_calls

                  )

                }

                return newMessages

              })

            } else if (

              chunk.event === RunEvent.RunResponse ||

              chunk.event === RunEvent.RunResponseContent

            ) {

              setMessages((*prevMessages*) => {

                const newMessages = [...prevMessages]

                const lastMessage = newMessages[newMessages.length - 1]

                if (

                  lastMessage &&

                  lastMessage.role === 'agent' &&

                  typeof chunk.content === 'string'

                ) {

                  const uniqueContent = chunk.content.replace(lastContent, '')

                  lastMessage.content += uniqueContent

                  lastContent = chunk.content

*// Handle tool calls streaming*

                  lastMessage.tool\_calls = processChunkToolCalls(

                    chunk,

                    lastMessage.tool\_calls

                  )

                  if (chunk.extra\_data?.reasoning\_steps) {

                    lastMessage.extra\_data = {

                      ...lastMessage.extra\_data,

                      reasoning\_steps: chunk.extra\_data.reasoning\_steps

                    }

                  }

                  if (chunk.extra\_data?.references) {

                    lastMessage.extra\_data = {

                      ...lastMessage.extra\_data,

                      references: chunk.extra\_data.references

                    }

                  }

                  lastMessage.created\_at =

                    chunk.created\_at ?? lastMessage.created\_at

                  if (chunk.images) {

                    lastMessage.images = chunk.images

                  }

                  if (chunk.videos) {

                    lastMessage.videos = chunk.videos

                  }

                  if (chunk.audio) {

                    lastMessage.audio = chunk.audio

                  }

                } else if (

                  lastMessage &&

                  lastMessage.role === 'agent' &&

                  typeof chunk?.content !== 'string' &&

                  chunk.content !== null

                ) {

                  const jsonBlock = getJsonMarkdown(chunk?.content)

                  lastMessage.content += jsonBlock

                  lastContent = jsonBlock

                } else if (

                  chunk.response\_audio?.transcript &&

                  typeof chunk.response\_audio?.transcript === 'string'

                ) {

                  const transcript = chunk.response\_audio.transcript

                  lastMessage.response\_audio = {

                    ...lastMessage.response\_audio,

                    transcript:

                      lastMessage.response\_audio?.transcript + transcript

                  }

                }

                return newMessages

              })

            } else if (chunk.event === RunEvent.ReasoningCompleted) {

              setMessages((*prevMessages*) => {

                const newMessages = [...prevMessages]

                const lastMessage = newMessages[newMessages.length - 1]

                if (lastMessage && lastMessage.role === 'agent') {

                  if (chunk.extra\_data?.reasoning\_steps) {

                    lastMessage.extra\_data = {

                      ...lastMessage.extra\_data,

                      reasoning\_steps: chunk.extra\_data.reasoning\_steps

                    }

                  }

                }

                return newMessages

              })

            } else if (chunk.event === RunEvent.RunError) {

              updateMessagesWithErrorState()

              const errorContent = chunk.content as string

              setStreamingErrorMessage(errorContent)

              if (hasStorage && newSessionId) {

                setSessionsData(

                  (*prevSessionsData*) =>

                    prevSessionsData?.filter(

                      (*session*) => session.session\_id !== newSessionId

                    ) ?? null

                )

              }

            } else if (chunk.event === RunEvent.RunCompleted) {

              setMessages((*prevMessages*) => {

                const newMessages = prevMessages.map((*message*, *index*) => {

                  if (

                    index === prevMessages.length - 1 &&

                    message.role === 'agent'

                  ) {

                    let updatedContent: string

                    if (typeof chunk.content === 'string') {

                      updatedContent = chunk.content

                    } else {

                      try {

                        updatedContent = JSON.stringify(chunk.content)

                      } catch {

                        updatedContent = 'Error parsing response'

                      }

                    }

                    return {

                      ...message,

                      content: updatedContent,

                      tool\_calls: processChunkToolCalls(

                        chunk,

                        message.tool\_calls

                      ),

                      images: chunk.images ?? message.images,

                      videos: chunk.videos ?? message.videos,

                      response\_audio: chunk.response\_audio,

                      created\_at: chunk.created\_at ?? message.created\_at,

                      extra\_data: {

                        reasoning\_steps:

                          chunk.extra\_data?.reasoning\_steps ??

                          message.extra\_data?.reasoning\_steps,

                        references:

                          chunk.extra\_data?.references ??

                          message.extra\_data?.references

                      }

                    }

                  }

                  return message

                })

                return newMessages

              })

            }

          },

          onError: (*error*) => {

            updateMessagesWithErrorState()

            setStreamingErrorMessage(error.message)

            if (hasStorage && newSessionId) {

              setSessionsData(

                (*prevSessionsData*) =>

                  prevSessionsData?.filter(

                    (*session*) => session.session\_id !== newSessionId

                  ) ?? null

              )

            }

          },

          onComplete: () => {}

        })

      } catch (error) {

        updateMessagesWithErrorState()

        setStreamingErrorMessage(

          error instanceof Error ? error.message : String(error)

        )

        if (hasStorage && newSessionId) {

          setSessionsData(

            (*prevSessionsData*) =>

              prevSessionsData?.filter(

                (*session*) => session.session\_id !== newSessionId

              ) ?? null

          )

        }

      } finally {

        focusChatInput()

        setIsStreaming(false)

      }

    },

    [

      setMessages,

      addMessage,

      updateMessagesWithErrorState,

      selectedEndpoint,

      streamResponse,

      agentId,

      setStreamingErrorMessage,

      setIsStreaming,

      focusChatInput,

      setSessionsData,

      sessionId,

      setSessionId,

      hasStorage,

      processChunkToolCalls

    ]

  )

  return { handleStreamResponse }

}

export default useAIChatStreamHandler

1. Tipado de datos (types/typings).



export interface ToolCall {

  role: 'user' | 'tool' | 'system' | 'assistant'

  content: string | null

  tool\_call\_id: string

  tool\_name: string

  tool\_args: Record<string, string>

  tool\_call\_error: boolean

  metrics: {

    time: number

  }

  created\_at: number

}

export interface ReasoningSteps {

  title: string

  action?: string

  result: string

  reasoning: string

  confidence?: number

  next\_action?: string

}

export interface ReasoningStepProps {

  index: number

  stepTitle: string

}

export interface ReasoningProps {

  reasoning: ReasoningSteps[]

}

export type ToolCallProps = {

  tools: ToolCall

}

interface ModelMessage {

  content: string | null

  context?: MessageContext[]

  created\_at: number

  metrics?: {

    time: number

    prompt\_tokens: number

    input\_tokens: number

    completion\_tokens: number

    output\_tokens: number

  }

  name: string | null

  role: string

  tool\_args?: unknown

  tool\_call\_id: string | null

  tool\_calls: Array<{

    function: {

      arguments: string

      name: string

    }

    id: string

    type: string

  }> | null

}

export interface Model {

  name: string

  model: string

  provider: string

}

export interface Agent {

  agent\_id: string

  name: string

  description: string

  model: Model

  storage?: boolean

}

interface MessageContext {

  query: string

  docs?: Array<Record<string, object>>

  time?: number

}

export enum RunEvent {

  RunStarted = 'RunStarted',

  RunResponse = 'RunResponse',

  RunResponseContent = 'RunResponseContent',

  RunCompleted = 'RunCompleted',

  RunError = 'RunError',

  ToolCallStarted = 'ToolCallStarted',

  ToolCallCompleted = 'ToolCallCompleted',

  UpdatingMemory = 'UpdatingMemory',

  ReasoningStarted = 'ReasoningStarted',

  ReasoningStep = 'ReasoningStep',

  ReasoningCompleted = 'ReasoningCompleted'

}

export interface ResponseAudio {

  id?: string

  content?: string

  transcript?: string

  channels?: number

  sample\_rate?: number

}

export interface NewRunResponse {

  status: 'RUNNING' | 'PAUSED' | 'CANCELLED'

}

export interface RunResponseContent {

  content?: string | object

  content\_type: string

  context?: MessageContext[]

  event: RunEvent

  event\_data?: object

  messages?: ModelMessage[]

  metrics?: object

  model?: string

  run\_id?: string

  agent\_id?: string

  session\_id?: string

  tool?: ToolCall

  tools?: Array<ToolCall>

  created\_at: number

  extra\_data?: PlaygroundAgentExtraData

  images?: ImageData[]

  videos?: VideoData[]

  audio?: AudioData[]

  response\_audio?: ResponseAudio

}

export interface RunResponse {

  content?: string | object

  content\_type: string

  context?: MessageContext[]

  event: RunEvent

  event\_data?: object

  messages?: ModelMessage[]

  metrics?: object

  model?: string

  run\_id?: string

  agent\_id?: string

  session\_id?: string

  tool?: ToolCall

  tools?: Array<ToolCall>

  created\_at: number

  extra\_data?: PlaygroundAgentExtraData

  images?: ImageData[]

  videos?: VideoData[]

  audio?: AudioData[]

  response\_audio?: ResponseAudio

}

export interface AgentExtraData {

  reasoning\_steps?: ReasoningSteps[]

  reasoning\_messages?: ReasoningMessage[]

  references?: ReferenceData[]

}

export interface PlaygroundAgentExtraData extends AgentExtraData {

  reasoning\_messages?: ReasoningMessage[]

  references?: ReferenceData[]

}

export interface ReasoningMessage {

  role: 'user' | 'tool' | 'system' | 'assistant'

  content: string | null

  tool\_call\_id?: string

  tool\_name?: string

  tool\_args?: Record<string, string>

  tool\_call\_error?: boolean

  metrics?: {

    time: number

  }

  created\_at?: number

}

export interface PlaygroundChatMessage {

  role: 'user' | 'agent' | 'system' | 'tool'

  content: string

  streamingError?: boolean

  created\_at: number

  tool\_calls?: ToolCall[]

  extra\_data?: {

    reasoning\_steps?: ReasoningSteps[]

    reasoning\_messages?: ReasoningMessage[]

    references?: ReferenceData[]

  }

  images?: ImageData[]

  videos?: VideoData[]

  audio?: AudioData[]

  response\_audio?: ResponseAudio

}

export interface ComboboxAgent {

  value: string

  label: string

  model: {

    provider: string

  }

  storage?: boolean

}

export interface ImageData {

  revised\_prompt: string

  url: string

}

export interface VideoData {

  id: number

  eta: number

  url: string

}

export interface AudioData {

  base64\_audio?: string

  mime\_type?: string

  url?: string

  id?: string

  content?: string

  channels?: number

  sample\_rate?: number

}

export interface ReferenceData {

  query: string

  references: Reference[]

  time?: number

}

export interface Reference {

  content: string

  meta\_data: {

    chunk: number

    chunk\_size: number

  }

  name: string

}

export interface SessionEntry {

  session\_id: string

  title: string

  created\_at: number

}

export interface ChatEntry {

  message: {

    role: 'user' | 'system' | 'tool' | 'assistant'

    content: string

    created\_at: number

  }

  response: {

    content: string

    tools?: ToolCall[]

    extra\_data?: {

      reasoning\_steps?: ReasoningSteps[]

      reasoning\_messages?: ReasoningMessage[]

      references?: ReferenceData[]

    }

    images?: ImageData[]

    videos?: VideoData[]

    audio?: AudioData[]

    response\_audio?: {

      transcript?: string

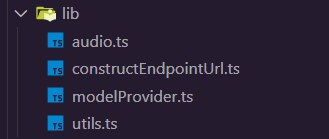
    }

    created\_at: number

  }

}

1. Funcionalidades de audio y utilidades relacionadas.



export function decodeBase64Audio(

*base64String*: string,

*mimeType* = 'audio/mpeg',

*sampleRate* = 44100,

*numChannels* = 1

): string {

*// Convert the Base64 string to binary*

  const byteString = atob(*base64String*)

  const byteArray = new Uint8Array(byteString.length)

  for (let i = 0; i < byteString.length; i += 1) {

    byteArray[i] = byteString.charCodeAt(i)

  }

  let blob: Blob

  if (*mimeType* === 'audio/pcm16') {

*// Convert PCM16 raw audio to WAV format*

    const wavHeader = createWavHeader(byteArray.length, *sampleRate*, *numChannels*)

    const wavData = new Uint8Array(wavHeader.length + byteArray.length)

    wavData.set(wavHeader, 0)

    wavData.set(byteArray, wavHeader.length)

    blob = new Blob([wavData], { type: 'audio/wav' }) *// Convert PCM to WAV*

  } else {

    blob = new Blob([byteArray], { type: *mimeType* })

  }

  return URL.createObjectURL(blob)

}

*// Function to generate WAV header for PCM16*

function createWavHeader(

*dataLength*: number,

*sampleRate*: number,

*numChannels*: number

): Uint8Array {

  const header = new ArrayBuffer(44)

  const view = new DataView(header)

  const blockAlign = *numChannels* \* 2 *// 16-bit PCM = 2 bytes per sample*

  const byteRate = *sampleRate* \* blockAlign

*// "RIFF" chunk descriptor*

  view.setUint32(0, 0x52494646, false) *// "RIFF"*

  view.setUint32(4, 36 + *dataLength*, true) *// File size*

  view.setUint32(8, 0x57415645, false) *// "WAVE"*

*// "fmt " sub-chunk*

  view.setUint32(12, 0x666d7420, false) *// "fmt "*

  view.setUint32(16, 16, true) *// Subchunk1 size*

  view.setUint16(20, 1, true) *// Audio format (1 = PCM)*

  view.setUint16(22, numChannels, true) *// Number of channels*

  view.setUint32(24, sampleRate, true) *// Sample rate*

  view.setUint32(28, byteRate, true) *// Byte rate*

  view.setUint16(32, blockAlign, true) *// Block align*

  view.setUint16(34, 16, true) *// Bits per sample (16-bit)*

*// "data" sub-chunk*

  view.setUint32(36, 0x64617461, false) *// "data"*

  view.setUint32(40, dataLength, true) *// Data size*

  return new Uint8Array(header)

}

import { clsx, type ClassValue } from 'clsx'

import { twMerge } from 'tailwind-merge'

export function cn(...*inputs*: ClassValue[]) {

  return twMerge(clsx(*inputs*))

}

export const truncateText = (*text*: string, *limit*: number) => {

  if (*text*) {

    return *text*.length > *limit* ? `${*text*.slice(0, *limit*)}..` : *text*

  }

  return ''

}

export const isValidUrl = (*url*: string): boolean => {

  try {

    const pattern = new RegExp(

      '^https?:\\/\\/' +

        '((([a-zA-Z\\d]([a-zA-Z\\d-]\*[a-zA-Z\\d])\*)\\.)+[a-zA-Z]{2,}|' +

        'localhost|' +

        '\\d{1,3}(\\.\\d{1,3}){3})' +

        '(\\:\\d+)?' +

        '(\\/[-a-zA-Z\\d%@\_.~+&:]\*)\*' +

        '(\\?[;&a-zA-Z\\d%@\_.,~+&:=-]\*)?' +

        '(\\#[-a-zA-Z\\d\_]\*)?$',

      'i'

    )

    return pattern.test(*url*.trim())

  } catch {

    return false

  }

}

export const getJsonMarkdown = (*content*: object = {}) => {

  let jsonBlock = ''

  try {

    jsonBlock = `\`\`\`json\n${JSON.stringify(*content*, null, 2)}\n\`\`\``

  } catch {

    jsonBlock = `\`\`\`\n${String(*content*)}\n\`\`\``

  }

  return jsonBlock

}