Chat App with FastAPI

Simple chat app example build with FastAPI.

Demonstrates:

- reusing chat history
- serializing messages
- streaming responses

This demonstrates storing chat history between requests and using it to give the model context for new responses.

Most of the complex logic here is between `chat_app.py` which streams the response to the browser, and `chat_app.ts` which renders messages in the browser.

Running the Example

With dependencies installed and environment variables set, run:

pip

```
python -m pydantic_ai_examples.chat_app
```

uv

```
uv run -m pydantic_ai_examples.chat_app
```

Then open the app at localhost:8000.

TODO screenshot.

Example Code

Python code that runs the chat app:

chat_app.py

```python
from __future__ import annotations as _annotations

import asyncio
import sqlite3
from collections.abc import AsyncIterator
from concurrent.futures.thread import ThreadPoolExecutor
from contextlib import asynccontextmanager
from dataclasses import dataclass
from functools import partial
from pathlib import Path
from typing import Annotated, Any, Callable, TypeVar

import fastapi
import logfire
from fastapi import Depends, Request
from fastapi.responses import HTMLResponse, Response, StreamingResponse
from pydantic import Field, TypeAdapter
from typing_extensions import LiteralString, ParamSpec

from pydantic_ai import Agent
from pydantic_ai.messages import (
    Message,
    MessagesTypeAdapter,
    ModelTextResponse,
    UserPrompt,
)

# 'if-token-present' means nothing will be sent (and the example will work) if you don't have logfire configured
logfire.configure(send_to_logfire='if-token-present')

agent = Agent('openai:gpt-4o')
THIS_DIR = Path(__file__).parent


@asynccontextmanager
async def lifespan(_app: fastapi.FastAPI):
    async with Database.connect() as db:
        yield {'db': db}


app = fastapi.FastAPI(lifespan=lifespan)
logfire.instrument_fastapi(app)


@app.get('/')
async def index() -> HTMLResponse:
    return HTMLResponse((THIS_DIR / 'chat_app.html').read_bytes())


@app.get('/chat_app.ts')
async def main_ts() -> Response:
    """Get the raw typescript code, it's compiled in the browser, forgive me."""
    return Response((THIS_DIR / 'chat_app.ts').read_bytes(), media_type='text/plain')


async def get_db(request: Request) -> Database:
    return request.state.db


@app.get('/chat/')
async def get_chat(database: Database = Depends(get_db)) -> Response:
    msgs = await database.get_messages()
    return Response(
        b'\n'.join(MessageTypeAdapter.dump_json(m) for m in msgs),
        media_type='text/plain',
    )


@app.post('/chat/')
```

```python
async def post_chat(
    prompt: Annotated[str, fastapi.Form()], database: Database = Depends(get_db)
) -> StreamingResponse:
    async def stream_messages():
        """Streams new line delimited JSON `Message`s to the client."""
        # stream the user prompt so that can be displayed straight away
        yield MessageTypeAdapter.dump_json(UserPrompt(content=prompt)) + b'\n'
        # get the chat history so far to pass as context to the agent
        messages = await database.get_messages()
        # run the agent with the user prompt and the chat history
        async with agent.run_stream(prompt, message_history=messages) as result:
            async for text in result.stream(debounce_by=0.01):
                # text here is a `str` and the frontend wants
                # JSON encoded ModelTextResponse, so we create one
                m = ModelTextResponse(content=text, timestamp=result.timestamp())
                yield MessageTypeAdapter.dump_json(m) + b'\n'

        # add new messages (e.g. the user prompt and the agent response in this case) to the database
        await database.add_messages(result.new_messages_json())

    return StreamingResponse(stream_messages(), media_type='text/plain')


MessageTypeAdapter: TypeAdapter[Message] = TypeAdapter(
    Annotated[Message, Field(discriminator='role')]
)
P = ParamSpec('P')
R = TypeVar('R')


@dataclass
class Database:
    """Rudimentary database to store chat messages in SQLite.

    The SQLite standard library package is synchronous, so we
    use a thread pool executor to run queries asynchronously.
    """

    con: sqlite3.Connection
    _loop: asyncio.AbstractEventLoop
    _executor: ThreadPoolExecutor

    @classmethod
    @asynccontextmanager
    async def connect(
        cls, file: Path = THIS_DIR / '.chat_app_messages.sqlite'
    ) -> AsyncIterator[Database]:
        with logfire.span('connect to DB'):
            loop = asyncio.get_event_loop()
            executor = ThreadPoolExecutor(max_workers=1)
            con = await loop.run_in_executor(executor, cls._connect, file)
            slf = cls(con, loop, executor)
        try:
            yield slf
        finally:
            await slf._asyncify(con.close)

    @staticmethod
    def _connect(file: Path) -> sqlite3.Connection:
        con = sqlite3.connect(str(file))
        con = logfire.instrument_sqlite3(con)
        cur = con.cursor()
        cur.execute(
            'CREATE TABLE IF NOT EXISTS messages (id INT PRIMARY KEY, message_list TEXT);'
        )
        con.commit()
        return con

    async def add_messages(self, messages: bytes):
        await self._asyncify(
            self._execute,
            'INSERT INTO messages (message_list) VALUES (?);',
            messages,
            commit=True,
        )
        await self._asyncify(self.con.commit)

    async def get_messages(self) -> list[Message]:
        c = await self._asyncify(
            self._execute, 'SELECT message_list FROM messages order by id desc'
        )
        rows = await self._asyncify(c.fetchall)
        messages: list[Message] = []
        for row in rows:
            messages.extend(MessagesTypeAdapter.validate_json(row[0]))
        return messages

    def _execute(
        self, sql: LiteralString, *args: Any, commit: bool = False
    ) -> sqlite3.Cursor:
        cur = self.con.cursor()
        cur.execute(sql, args)
        if commit:
            self.con.commit()
        return cur

    async def _asyncify(
        self, func: Callable[P, R], *args: P.args, **kwargs: P.kwargs
    ) -> R:
        return await self._loop.run_in_executor(  # type: ignore
            self._executor,
            partial(func, **kwargs),
            *args,  # type: ignore
        )


if __name__ == '__main__':
    import uvicorn

    uvicorn.run(
        'pydantic_ai_examples.chat_app:app', reload=True, reload_dirs=[str(THIS_DIR)]
    )
```

Simple HTML page to render the app:

chat_app.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Chat App</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
  <style>
    main {
      max-width: 700px;
    }
    #conversation .user::before {
      content: 'You asked: ';
      font-weight: bold;
      display: block;
    }
    #conversation .llm-response::before {
      content: 'AI Response: ';
      font-weight: bold;
      display: block;
    }
    #spinner {
      opacity: 0;
      transition: opacity 500ms ease-in;
      width: 30px;
      height: 30px;
      border: 3px solid #222;
      border-bottom-color: transparent;
      border-radius: 50%;
      animation: rotation 1s linear infinite;
    }
    @keyframes rotation {
      0% { transform: rotate(0deg); }
      100% { transform: rotate(360deg); }
    }
    #spinner.active {
      opacity: 1;
    }
  </style>
</head>
<body>
  <main class="border rounded mx-auto my-5 p-4">
    <h1>Chat App</h1>
    <p>Ask me anything...</p>
    <div id="conversation" class="px-2"></div>
    <div class="d-flex justify-content-center mb-3">
      <div id="spinner"></div>
    </div>
    <form method="post">
      <input id="prompt-input" name="prompt" class="form-control"/>
      <div class="d-flex justify-content-end">
        <button class="btn btn-primary mt-2">Send</button>
      </div>
    </form>
    <div id="error" class="d-none text-danger">
      Error occurred, check the console for more information.
    </div>
  </main>
</body>
</html>
<script src="https://cdnjs.cloudflare.com/ajax/libs/typescript/5.6.3/typescript.min.js" crossorigin="anonymous" referrerpolicy="no-referrer"></script>
<script type="module">
  // to let me write TypeScript, without adding the burden of npm we do a dirty, non-production-ready hack
  // and transpile the TypeScript code in the browser
  // this is (arguably) A neat demo trick, but not suitable for production!
  async function loadTs() {
    const response = await fetch('/chat_app.ts');
    const tsCode = await response.text();
    const jsCode = window.ts.transpile(tsCode, { target: "es2015" });
    let script = document.createElement('script');
    script.type = 'module';
    script.text = jsCode;
    document.body.appendChild(script);
  }

  loadTs().catch((e) => {
    console.error(e);
    document.getElementById('error').classList.remove('d-none');
    document.getElementById('spinner').classList.remove('active');
  });
</script>
```

TypeScript to handle rendering the messages, to keep this simple (and at the risk of offending frontend developers) the typescript code is passed to the browser as plain text and transpiled in the browser.

chat_app.ts

```
// BIG FAT WARNING: to avoid the complexity of npm, this typescript is compiled in the browser
// there's currently no static type checking

import { marked } from 'https://cdnjs.cloudflare.com/ajax/libs/marked/15.0.0/lib/marked.esm.js'
const convElement = document.getElementById('conversation')

const promptInput = document.getElementById('prompt-input') as HTMLInputElement
const spinner = document.getElementById('spinner')

// stream the response and render messages as each chunk is received
// data is sent as newline-delimited JSON
async function onFetchResponse(response: Response): Promise<void> {
  let text = ''
  let decoder = new TextDecoder()
  if (response.ok) {
    const reader = response.body.getReader()
    while (true) {
      const {done, value} = await reader.read()
      if (done) {
        break
      }
      text += decoder.decode(value)
      addMessages(text)
      spinner.classList.remove('active')
    }
    addMessages(text)
    promptInput.disabled = false
    promptInput.focus()
  } else {
    const text = await response.text()
    console.error(`Unexpected response: ${response.status}`, {response, text})
    throw new Error(`Unexpected response: ${response.status}`)
  }
}

// The format of messages, this matches pydantic-ai both for brevity and understanding
// in production, you might not want to keep this format all the way to the frontend
interface Message {
  role: string
```

```ts
  content: string
  timestamp: string
}

// take raw response text and render messages into the `#conversation` element
// Message timestamp is assumed to be a unique identifier of a message, and is used to deduplicate
// hence you can send data about the same message multiple times, and it will be updated
// instead of creating a new message elements
function addMessages(responseText: string) {
  const lines = responseText.split('\n')
  const messages: Message[] = lines.filter(line => line.length > 1).map(j => JSON.parse(j))
  for (const message of messages) {
    // we use the timestamp as a crude element id
    const {timestamp, role, content} = message
    const id = `msg-${timestamp}`
    let msgDiv = document.getElementById(id)
    if (!msgDiv) {
      msgDiv = document.createElement('div')
      msgDiv.id = id
      msgDiv.title = `${role} at ${timestamp}`
      msgDiv.classList.add('border-top', 'pt-2', role)
      convElement.appendChild(msgDiv)
    }
    msgDiv.innerHTML = marked.parse(content)
  }
  window.scrollTo({ top: document.body.scrollHeight, behavior: 'smooth' })
}

function onError(error: any) {
  console.error(error)
  document.getElementById('error').classList.remove('d-none')
  document.getElementById('spinner').classList.remove('active')
}

async function onSubmit(e: SubmitEvent): Promise<void> {
  e.preventDefault()
  spinner.classList.add('active')
  const body = new FormData(e.target as HTMLFormElement)

  promptInput.value = ''
  promptInput.disabled = true

  const response = await fetch('/chat/', {method: 'POST', body})
  await onFetchResponse(response)
}

// call onSubmit when the form is submitted (e.g. user clicks the send button or hits Enter)
document.querySelector('form').addEventListener('submit', (e) => onSubmit(e).catch(onError))

// load messages on page load
fetch('/chat/').then(onFetchResponse).catch(onError)
```