TechDocsChatService

September 4, 2026

TechDocsChatService

Kind: Service

Source: `atloria-monorepo/apps/api/src/technical-docs/rag/techdocs-chat.service.ts`

Foundation F3: conversational grounded chat — sessions, memory, streaming, feedback.
Wraps the one-shot TechDocsRagService with per-session history so follow-up questions
("what about its retry behavior?") resolve against the conversation, and records answered/refused/feedback signals into the F2 event stream for the deflection dashboard.

`TechDocsChatService` provides session-based, grounded conversational chat over technical documentation. It wraps the one-shot `TechDocsRagService` by maintaining per-session message history (memory) so follow-up questions resolve in context, supports streaming responses, and records answered/refused/feedback signals into the F2 event stream for deflection analytics.

Methods

Method	Signature	Returns	Description
`createSession`	`createSession(projectId: string, organizationId: string	null, channel: string, userId: string	null)`
`askStream`	`askStream(projectId: string, sessionId: string, question: string)`	`AsyncGenerator<string, {	

```
messageId: string;
citations: string[];
sources: DocSearchHit[];
usage: { inputTokens: number; outputTokens: number; totalTokens: number };
```

}, void> | Streamed, grounded, session-aware ask. | | escalate | escalate(projectId: string, sessionId: string, dto: { email?: string; note?: string; pageUrl?: string }) | unknown | 3.1 deflection loop: escalate a chat session to a human. |
| feedback | feedback(projectId: string, messageId: string, rating: 'up' | 'down', comment: string) | unknown` | 👍/👎 on an assistant message → message row + F2 event (deflection dashboard input). |

Dependencies

- `PrismaService`
- `TechDocsRagService`
- `RagChatModelService`
- `EventsService`
- `IssuesService`

Where it refuses work

- `TechDocsChatService` stops the work with `NotFoundException` when `!session || session.projectId !== projectId` — “Chat session not found.”, in 2 places.
- `TechDocsChatService` stops the work with `BadRequestException` when `!question.trim()` — “A question is required.”.
- `TechDocsChatService` stops the work with `BadRequestException` when `note.length > 2000` — “note too long (max 2000 chars)”.
- `TechDocsChatService` stops the work with `BadRequestException` when `rating !== 'up' && rating !== 'down'` — “rating must be "up" or "down"”.
- `TechDocsChatService` stops the work with `NotFoundException` when `!msg || msg.session.projectId !== projectId` — “Message not found.”.

When something fails

- `TechDocsChatService` handles failure in 1 place: it logs it and continues in all 1.

Diagram

```
sequenceDiagram
  autonumber
  actor User
  participant API as Controller/Resolver
  participant Chat as TechDocsChatService
```

```
  participant Store as Session Memory Store
  participant RAG as TechDocsRagService
  participant F2 as F2 Event Stream

  User->>API: sendMessage(sessionId, prompt)
  API->>Chat: chat(sessionId, prompt, opts)
  Chat->>Store: loadHistory(sessionId)
  Store-->>Chat: history[]
  Chat->>RAG: ask({ prompt, history, streaming: true })
  RAG-->>Chat: stream tokens / final answer
  Chat-->>API: stream to client
  Chat->>Store: appendTurn(sessionId, user+assistant)
  Chat->>F2: emit(answered|refused, metadata)
  User->>API: submitFeedback(sessionId, messageId, vote)
  API->>Chat: recordFeedback(...)
  Chat->>F2: emit(feedback, metadata)
```

Usage

```
import { Injectable } from '@nestjs/common';
import { TechDocsChatService } from './technical-docs/rag/techdocs-chat.service';

@Injectable()
export class TechDocsChatController {
  constructor(private readonly chat: TechDocsChatService) {}

  // Pseudo-handler; adapt to your transport (REST/GraphQL/WebSocket/SSE)
  async ask(sessionId: string, prompt: string) {
    // Non-streaming example (if supported by your method signature)
    const result = await this.chat.chat({
      sessionId,
      message: prompt,
      // optional: userId/tenantId/document scope/etc. depending on your app
    });

    return {
      sessionId,
      answer: result.answer,
      refused: result.refused,
      citations: result.citations,
    };
  }

  // Streaming example (SSE/WebSocket): consume async iterator/callback stream
  async askStream(sessionId: string, prompt: string, onToken: (t: string) => void) {
    const stream = await this.chat.chatStream({ sessionId, message: prompt });

    for await (const chunk of stream) {
      // chunk could be token/text delta depending on implementation
      onToken(chunk.text ?? String(chunk));
```

```
    }
  }

  async feedback(sessionId: string, messageId: string, vote: 'up' | 'down') {
    await this.chat.recordFeedback({ sessionId, messageId, vote });
    return { ok: true };
  }
}
```

AI Coding Instructions

- Preserve the “one-shot RAG + session memory” layering: `TechDocsChatService` should manage history/streaming/telemetry, while `TechDocsRagService` stays focused on retrieval + answer generation.
- Always load and append conversation turns atomically per `sessionId` to avoid interleaving history when multiple requests stream concurrently.
- Emit consistent F2 events for answered/refused and feedback, including stable identifiers (sessionId, messageId, doc scope) so the deflection dashboard can aggregate correctly.
- When adding new options (e.g., doc filters, tenant scoping), pass them through to the RAG call and include them in event metadata; avoid storing large retrieved context blobs in session memory.

Relationships

- DEPENDS_ON → `PrismaService`
- DEPENDS_ON → `TechDocsRagService`
- DEPENDS_ON → `RagChatModelService`
- DEPENDS_ON → `EventsService`
- DEPENDS_ON → `IssuesService`

Referenced By

- `TechnicalDocsChatController` (DEPENDS_ON)
- `TechnicalDocsModule` (MODULE_PROVIDES)