TechnicalDocsEnrichmentService

September 4, 2026

TechnicalDocsEnrichmentService

Kind: Service

Source: `atloria-monorepo/apps/api/src/technical-docs/technical-docs-enrichment.service.ts`

`TechnicalDocsEnrichmentService` generates AI-powered documentation for parsed knowledge-graph entities, including missing descriptions, Mermaid diagrams, usage examples, and coding guidance. It runs asynchronously after `TechnicalDocsParserService` saves a snapshot, persisting each result as an `AI_ENRICHMENT` `Document` keyed by a deterministic ID so enrichments survive subsequent parses.

Methods

Method	Signature	Returns	Description
`enrichProject`	`enrichProject(projectId: string, organizationId: string, userId: string, limit: number)`	`Promise<{ entities: number; enriched: number }>`	Enrich all entities from a snapshot.
`enrichSnapshot`	`enrichSnapshot(projectId: string, organizationId: string, entities: Entity[], userId: string, limit: number, relationships: Relationship[])`	`Promise<number>`	
`describeSubsystem`	`describeSubsystem(input: {		

```
name: string;
path: string;
entityCount: number;
typeBreakdown: string;
entryPoints: string[];
dependents: string[];
dependsOn: string[];
sampleNames: string[];
```

```
/**
 * What the developers themselves wrote — the first sentence of each member's doc comment.
 *
 * Bounded by CHARACTERS, not item count. A first attempt capped this at 12 comments, which on
 * a large subsystem is several thousand characters; the model reasoned proportionally and
 * spent its whole budget before emitting a word. Twelve short comments and twelve long ones
 * are not the same prompt.
 */
docComments?: string[];
/** Conditions this subsystem refuses work on, with the developer's own message. */
refusals?: string[];
/** The ordered call chain traced through resolved dependency edges. */
flow?: string;
```

}) | Promise<string | null>` | Build a concise summary of the entity for the AI prompt. |

Dependencies

- `PrismaService`
- `AIProvider` *(optional)*
- `AIUsageService` *(optional)*

Where it refuses work

- `TechnicalDocsEnrichmentService` stops the work with an early return when `!this.buildEntitySummary(entity)`.
- `TechnicalDocsEnrichmentService` stops the work with an early return when `!entitySummary`.
- `TechnicalDocsEnrichmentService` stops the work with an early return when `!rawContent`.
- `TechnicalDocsEnrichmentService` stops the work with an early return when `!this.aiProvider`.
- `TechnicalDocsEnrichmentService` stops the work with an early return when `!hasContent`.
- `TechnicalDocsEnrichmentService` stops the work with an early return when `!raw?.trim()`.

When something fails

- `TechnicalDocsEnrichmentService` handles failure in 3 places: it turns it into a return value in 2, and logs it and continues in 1.

Diagram

```
sequenceDiagram
    participant Parser as TechnicalDocsParserService
    participant Enrichment as TechnicalDocsEnrichmentService
    participant AI as AI Provider
    participant DB as Document Repository

    Parser->>DB: Save parsed entity snapshot
    Parser-->>Enrichment: Trigger enrichment asynchronously
    Enrichment->>Enrichment: Identify missing/thin metadata
    Enrichment->>AI: Generate description, diagram, examples, instructions
    AI-->>Enrichment: Return generated documentation
    Enrichment->>DB: Upsert AI_ENRICHMENT Documents
    Note over DB: Deterministic Document.id preserves<br/>enrichments across re-parses
```

Usage

```
import { Injectable } from '@nestjs/common';
import { TechnicalDocsEnrichmentService } from './technical-docs-enrichment.service';

@Injectable()
export class TechnicalDocsParserService {
  constructor(
    private readonly enrichmentService: TechnicalDocsEnrichmentService,
  ) {}

  async parseAndSave(sourcePath: string) {
    const snapshot = await this.parseSource(sourcePath);
    const entities = await this.saveSnapshot(snapshot);

    // Do not block parsing or snapshot persistence on AI generation.
    void Promise.allSettled(
      entities.map((entity) =>
        this.enrichmentService.enrichEntity(entity),
      ),
    );

    return entities;
  }

  private async parseSource(sourcePath: string) {
    // Parse source files into a technical-documentation snapshot.
  }

  private async saveSnapshot(snapshot: unknown) {
    // Persist entities and return the saved entity records.
    return [];
```

```
  }
}
```

AI Coding Instructions

- Trigger enrichment only after the parsed snapshot and its entities have been persisted; enrichment documents require a stable `kgEntityId`.
- Keep enrichment non-blocking (`void` or a background job) so AI provider latency or failures never prevent parsing from completing.
- Persist generated output as `Document` records with `source: 'AI'`, `type: 'AI_ENRICHMENT'`, and the associated `kgEntityId`.
- Use deterministic `Document.id` values when upserting enrichments to preserve AI-generated content across source re-parses.
- Generate diagrams appropriate to the entity type: sequence diagrams for services, ER diagrams for models, and component hierarchies for structural entities.

Relationships

- DEPENDS_ON → `PrismaService`
- DEPENDS_ON → `aiprovider`
- DEPENDS_ON → `AIUsageService`

Referenced By

- `TechnicalDocsMaterializerService` (DEPENDS_ON)
- `TechnicalDocsParserService` (DEPENDS_ON)
- `TechnicalDocsQueue` (DEPENDS_ON)
- `TechnicalDocsModule` (MODULE_PROVIDES)
- `TechnicalDocsModule` (MODULE_EXPORTS)