# RAG

RAG search example. This demo allows you to ask question of the logfire documentation.

Demonstrates:

- tools
- agent dependencies
- RAG search

This is done by creating a database containing each section of the markdown documentation, then registering the search tool with the PydanticAI agent.

Logic for extracting sections from markdown files and a JSON file with that data is available in this gist.

PostgreSQL with pgvector is used as the search database, the easiest way to download and run pgvector is using Docker:

```
mkdir postgres-data
docker run --rm \
  -e POSTGRES_PASSWORD=postgres \
  -p 54320:5432 \
  -v `pwd`/postgres-data:/var/lib/postgresql/data \
  pgvector/pgvector:pg17
```

As with the SQL gen example, we run postgres on port `54320` to avoid conflicts with any other postgres instances you may have running. We also mount the PostgreSQL `data` directory locally to persist the data if you need to stop and restart the container.

With that running and dependencies installed and environment variables set, we can build the search database with (**WARNING**: this requires the `OPENAI_API_KEY` env variable and will calling the OpenAI embedding API around 300 times to generate embeddings for each section of the documentation):

**pip**

```
python -m pydantic_ai_examples.rag build
```

**uv**

```
uv run -m pydantic_ai_examples.rag build
```

(Note building the database doesn't use PydanticAI right now, instead it uses the OpenAI SDK directly.)

You can then ask the agent a question with:

**pip**

```
python -m pydantic_ai_examples.rag search "How do I configure logfire to work with FastAPI?"
```

**uv**

```
uv run -m pydantic_ai_examples.rag search "How do I configure logfire to work with FastAPI?"
```

## Example Code

**rag.py**

```python
from __future__ import annotations as _annotations

import asyncio
import re
import sys
import unicodedata
from contextlib import asynccontextmanager
from dataclasses import dataclass

import asyncpg
import httpx
import logfire
import pydantic_core
from openai import AsyncOpenAI
from pydantic import TypeAdapter
from typing_extensions import AsyncGenerator

from pydantic_ai import RunContext
from pydantic_ai.agent import Agent

# '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')
logfire.instrument_asyncpg()


@dataclass
class Deps:
    openai: AsyncOpenAI
    pool: asyncpg.Pool


agent = Agent('openai:gpt-4o', deps_type=Deps)


@agent.tool
async def retrieve(context: RunContext[Deps], search_query: str) -> str:
    """Retrieve documentation sections based on a search query.

    Args:
        context: The call context.
        search_query: The search query.
    """
    with logfire.span(
        'create embedding for {search_query=}', search_query=search_query
    ):
        embedding = await context.deps.openai.embeddings.create(
            input=search_query,
            model='text-embedding-3-small',
        )
```

```
    assert (
        len(embedding.data) == 1
    ), f'Expected 1 embedding, got {len(embedding.data)}, doc query: {search_query!r}'
    embedding = embedding.data[0].embedding
    embedding_json = pydantic_core.to_json(embedding).decode()
    rows = await context.deps.pool.fetch(
        'SELECT url, title, content FROM doc_sections ORDER BY embedding <-> $1 LIMIT 8',
        embedding_json,
    )
    return '\n\n'.join(
        f'# {row["title"]}\nDocumentation URL:{row["url"]}\n\n{row["content"]}\n'
        for row in rows
    )


async def run_agent(question: str):
    """Entry point to run the agent and perform RAG based question answering."""
    openai = AsyncOpenAI()
    logfire.instrument_openai(openai)

    logfire.info('Asking "{question}"', question=question)

    async with database_connect(False) as pool:
        deps = Deps(openai=openai, pool=pool)
        answer = await agent.run(question, deps=deps)
    print(answer.data)


#######################################################
# The rest of this file is dedicated to preparing the #
# search database, and some utilities.                #
#######################################################

# JSON document from
# https://gist.github.com/samuelcolvin/4b5bb9bb163b1122ff17e29e48c10992
DOCS_JSON = (
    'https://gist.githubusercontent.com/'
    'samuelcolvin/4b5bb9bb163b1122ff17e29e48c10992/raw/'
    '80c5925c42f1442c24963aaf5eb1a324d47afe95/logfire_docs.json'
)


async def build_search_db():
    """Build the search database."""
    async with httpx.AsyncClient() as client:
        response = await client.get(DOCS_JSON)
        response.raise_for_status()
    sections = sessions_ta.validate_json(response.content)

    openai = AsyncOpenAI()
    logfire.instrument_openai(openai)

    async with database_connect(True) as pool:
        with logfire.span('create schema'):
            async with pool.acquire() as conn:
                async with conn.transaction():
                    await conn.execute(DB_SCHEMA)

        sem = asyncio.Semaphore(10)
        async with asyncio.TaskGroup() as tg:
            for section in sections:
                tg.create_task(insert_doc_section(sem, openai, pool, section))


async def insert_doc_section(
    sem: asyncio.Semaphore,
    openai: AsyncOpenAI,
    pool: asyncpg.Pool,
    section: DocsSection,
) -> None:
    async with sem:
        url = section.url()
        exists = await pool.fetchval('SELECT 1 FROM doc_sections WHERE url = $1', url)
        if exists:
            logfire.info('Skipping {url=}', url=url)

        with logfire.span('create embedding for {url=}', url=url):
            embedding = await openai.embeddings.create(
                input=section.embedding_content(),
                model='text-embedding-3-small',
            )
        assert (
            len(embedding.data) == 1
        ), f'Expected 1 embedding, got {len(embedding.data)}, doc section: {section}'
        embedding = embedding.data[0].embedding
        embedding_json = pydantic_core.to_json(embedding).decode()
        await pool.execute(
            'INSERT INTO doc_sections (url, title, content, embedding) VALUES ($1, $2, $3, $4)',
            url,
            section.title,
            section.content,
            embedding_json,
        )


@dataclass
class DocsSection:
    id: int
    parent: int | None
    path: str
    level: int
    title: str
    content: str

    def url(self) -> str:
        url_path = re.sub(r'\.md$', '', self.path)
        return (
            f'https://logfire.pydantic.dev/docs/{url_path}/#{slugify(self.title, "-")}'
        )

    def embedding_content(self) -> str:
        return '\n\n'.join((f'path: {self.path}', f'title: {self.title}', self.content))


sessions_ta = TypeAdapter(list[DocsSection])


# pyright: reportUnknownMemberType=false
# pyright: reportUnknownVariableType=false
@asynccontextmanager
async def database_connect(
    create_db: bool = False,
) -> AsyncGenerator[asyncpg.Pool, None]:
```

```
    server_dsn, database = (
        'postgresql://postgres:postgres@localhost:54320',
        'pydantic_ai_rag',
    )
    if create_db:
        with logfire.span('check and create DB'):
            conn = await asyncpg.connect(server_dsn)
            try:
                db_exists = await conn.fetchval(
                    'SELECT 1 FROM pg_database WHERE datname = $1', database
                )
                if not db_exists:
                    await conn.execute(f'CREATE DATABASE {database}')
            finally:
                await conn.close()

    pool = await asyncpg.create_pool(f'{server_dsn}/{database}')
    try:
        yield pool
    finally:
        await pool.close()


DB_SCHEMA = """
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE IF NOT EXISTS doc_sections (
    id serial PRIMARY KEY,
    url text NOT NULL UNIQUE,
    title text NOT NULL,
    content text NOT NULL,
    -- text-embedding-3-small returns a vector of 1536 floats
    embedding vector(1536) NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_doc_sections_embedding ON doc_sections USING hnsw (embedding vector_l2_ops);
"""


def slugify(value: str, separator: str, unicode: bool = False) -> str:
    """Slugify a string, to make it URL friendly."""
    # Taken unchanged from https://github.com/Python-Markdown/markdown/blob/3.7/markdown/extensions/toc.py#L38
    if not unicode:
        # Replace Extended Latin characters with ASCII, i.e. `žlutý` => `zluty`
        value = unicodedata.normalize('NFKD', value)
        value = value.encode('ascii', 'ignore').decode('ascii')
    value = re.sub(r'[^\w\s-]', '', value).strip().lower()
    return re.sub(rf'[{separator}\s]+', separator, value)


if __name__ == '__main__':
    action = sys.argv[1] if len(sys.argv) > 1 else None
    if action == 'build':
        asyncio.run(build_search_db())
    elif action == 'search':
        if len(sys.argv) == 3:
            q = sys.argv[2]
        else:
            q = 'How do I configure logfire to work with FastAPI?'
        asyncio.run(run_agent(q))
    else:
        print(
            'uv run --extra examples -m pydantic_ai_examples.rag build|search',
            file=sys.stderr,
        )
        sys.exit(1)
```