SQL Generation

Example demonstrating how to use PydanticAI to generate SQL queries based on user input.

Demonstrates:

- dynamic system prompt
- structured `result_type`
- result validation
- agent dependencies

Running the Example

The resulting SQL is validated by running it as an `EXPLAIN` query on PostgreSQL. To run the example, you first need to run PostgreSQL, e.g. via Docker:

```
docker run --rm -e POSTGRES_PASSWORD=postgres -p 54320:5432 postgres
```

(we run postgres on port `54320` to avoid conflicts with any other postgres instances you may have running)

With dependencies installed and environment variables set, run:

pip

```
python -m pydantic_ai_examples.sql_gen
```

uv

```
uv run -m pydantic_ai_examples.sql_gen
```

or to use a custom prompt:

pip

```
python -m pydantic_ai_examples.sql_gen "find me errors"
```

uv

```
uv run -m pydantic_ai_examples.sql_gen "find me errors"
```

This model uses `gemini-1.5-flash` by default since Gemini is good at single shot queries of this kind.

Example Code

sql_gen.py

```python
import asyncio
import sys
from collections.abc import AsyncGenerator
from contextlib import asynccontextmanager
from dataclasses import dataclass
from datetime import date
from typing import Annotated, Any, Union

import asyncpg
import logfire
from annotated_types import MinLen
from devtools import debug
from pydantic import BaseModel, Field
from typing_extensions import TypeAlias

from pydantic_ai import Agent, ModelRetry, RunContext

# '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()

DB_SCHEMA = """
CREATE TABLE records (
    created_at timestamptz,
    start_timestamp timestamptz,
    end_timestamp timestamptz,
    trace_id text,
    span_id text,
    parent_span_id text,
    level log_level,
    span_name text,
    message text,
    attributes_json_schema text,
    attributes jsonb,
    tags text[],
    is_exception boolean,
    otel_status_message text,
    service_name text
);
"""


@dataclass
class Deps:
    conn: asyncpg.Connection


class Success(BaseModel):
    """Response when SQL could be successfully generated."""

    sql_query: Annotated[str, MinLen(1)]
    explanation: str = Field(
        '', description='Explanation of the SQL query, as markdown'
    )


class InvalidRequest(BaseModel):
```

```
    """Response the user input didn't include enough information to generate SQL."""

    error_message: str


Response: TypeAlias = Union[Success, InvalidRequest]
agent = Agent(
    'gemini-1.5-flash',
    # Type ignore while we wait for PEP-0747, nonetheless unions will work fine everywhere else
    result_type=Response,  # type: ignore
    deps_type=Deps,
)


@agent.system_prompt
async def system_prompt() -> str:
    return f"""\
Given the following PostgreSQL table of records, your job is to
write a SQL query that suits the user's request.

Database schema:

{DB_SCHEMA}

today's date = {date.today()}

Example
    request: show me records where foobar is false
    response: SELECT * FROM records WHERE attributes->>'foobar' = false
Example
    request: show me records where attributes include the key "foobar"
    response: SELECT * FROM records WHERE attributes ? 'foobar'
Example
    request: show me records from yesterday
    response: SELECT * FROM records WHERE start_timestamp::date > CURRENT_TIMESTAMP - INTERVAL '1 day'
Example
    request: show me error records with the tag "foobar"
    response: SELECT * FROM records WHERE level = 'error' and 'foobar' = ANY(tags)
"""


@agent.result_validator
async def validate_result(ctx: RunContext[Deps], result: Response) -> Response:
    if isinstance(result, InvalidRequest):
        return result

    # gemini often adds extraneous backslashes to SQL
    result.sql_query = result.sql_query.replace('\\', '')
    if not result.sql_query.upper().startswith('SELECT'):
        raise ModelRetry('Please create a SELECT query')

    try:
        await ctx.deps.conn.execute(f'EXPLAIN {result.sql_query}')
    except asyncpg.exceptions.PostgresError as e:
        raise ModelRetry(f'Invalid query: {e}') from e
    else:
        return result


async def main():
    if len(sys.argv) == 1:
        prompt = 'show me logs from yesterday, with level "error"'
    else:
        prompt = sys.argv[1]

    async with database_connect(
        'postgresql://postgres:postgres@localhost:54320', 'pydantic_ai_sql_gen'
    ) as conn:
        deps = Deps(conn)
        result = await agent.run(prompt, deps=deps)
    debug(result.data)


# pyright: reportUnknownMemberType=false
# pyright: reportUnknownVariableType=false
@asynccontextmanager
async def database_connect(server_dsn: str, database: str) -> AsyncGenerator[Any, None]:
    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()

    conn = await asyncpg.connect(f'{server_dsn}/{database}')
    try:
        with logfire.span('create schema'):
            async with conn.transaction():
                if not db_exists:
                    await conn.execute(
                        "CREATE TYPE log_level AS ENUM ('debug', 'info', 'warning', 'error', 'critical')"
                    )
                await conn.execute(DB_SCHEMA)
        yield conn
    finally:
        await conn.close()


if __name__ == '__main__':
    asyncio.run(main())
```