
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

2. `SQL`

1) `SQL`

- 주요 개요
  - a. **`create_sql_query_chain`** 을 활용 → `chain` 생성 → `SQL` 쿼리 생성 & 실행 → 답변 도출
  - b. **`SQL Agent`**

- **`환경설정`**

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv

# API 키 정보 로드
load_dotenv()                                # True
```

---

2) `create_sql_query_chain`

- **`SQL Database 정보 로드하기`**

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.chains import create_sql_query_chain
from langchain_community.utilities import SQLDatabase

# SQLite 데이터베이스에 연결하기
db = SQLDatabase.from_uri("sqlite:///data/finance.db")

# 데이터베이스의 사용 연결 확인하기
print(db.dialect)

# 사용 가능한 테이블 이름들을 출력하기
```

```
print(db.get_usable_table_names())

# 데이터베이스 연결 확인하기
print("데이터베이스 연결 완료")
```

- `sqlite`

```
['accounts', 'customers', 'transactions']
```

- 데이터베이스 연결 완료

- **`LLM` 객체 생성 → `LLM` + `DB`를 매개변수로 입력 → `chain` 생성하기**

```
# LLM 생성하기

import os
from dotenv import load_dotenv

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-lite",
    temperature=0,                                          # temperature = 0
    max_output_tokens=4096,
)
```

- `gemini-2.5-flash-lite` 생성하기

```
E0000 00:00:1760243592.183319 4613744 alts_credentials.cc:93] ALTS creds ignore
```

```
# LLM + DB 를 매개변수로 입력 → chain을 생성하기

chain = create_sql_query_chain(gemini_lc, db)
```

- *(Optional) 아래의 방식으로 `Prompt`를 직접 지정 가능*
  - *직접 작성시 `table_info`와 더불어 설명가능한 `column description`을 추가할 수 있음*

```
from langchain_core.prompts import PromptTemplate

# 프롬프트 직접 지정하기
```

```
prompt = PromptTemplate.from_template(
    """Given an input question, first create a syntactically correct {dialect} qu
Use the following format:

Question: "Question here"
SQLQuery: "SQL Query to run"
SQLResult: "Result of the SQLQuery"
Answer: "Final answer here"

Only use the following tables:
{table_info}

Here is the description of the columns in the tables:
`cust`: customer name
`prod`: product name
`trans`: transaction date

Question: {input}"""
).partial(dialect=db.dialect)
```

```
# LLM + DB 를 매개변수로 입력 → chain을 생성하기

chain_2 = create_sql_query_chain(gemini_lc, db, prompt)
```

- **`chain_2` 실행 → `DB` 기반 → `쿼리` 생성하기**

```
# chain_2 실행 → 결과 출력해보기

generated_sql_query = chain_2.invoke({"question": "고객의 이름을 나열하세요"})
```

```
# 생성된 쿼리 출력하기

print(generated_sql_query.__repr__())
```

- **`'SELECT name FROM customers'`** - (`0.8s`)

- **`생성한 쿼리가 맞게 동작하는지 확인해보기`**

```
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool

# 생성한 쿼리를 실행하기 위한 도구를 생성해보기
execute_query = QuerySQLDataBaseTool(db=db)
```

- `생성한 쿼리를 실행하기 위한 도구 생성하기`

```
/var/folders/h3/l7wnkv352kqftv0t8ctl2ld40000gn/T/ipykernel_87702/45556421.py:4:
```

```
execute_query = QuerySQLDataBaseTool(db=db)
```

```
execute_query.invoke({"query": generated_sql_query})
```

- **생한 쿼리 실행하기 위한 도구 생성**

```
"[('테디',), ('폴',), ('셜리',), ('민수',), ('지영',), ('은정',)]"
```

```
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool

# 도구
execute_query = QuerySQLDataBaseTool(db=db)

# SQL 쿼리 생성 체인
write_query = create_sql_query_chain(gemini_lc, db)

# 생성한 쿼리를 실행하기 위한 체인을 생성하기
chain_3 = write_query | execute_query
```

```
# 실행 결과 확인하기

chain_3.invoke({"question": "테디의 이메일을 조회하세요"})
```

- **"[('teddy@example.com',)]"** - (0.8s)

---

2) 답변을 LLM으로 증강-생성

- 이전 단계에서 생성한 chain을 사용 → 답변 = 단답형 형식으로 출력
- 이를 **LCEL** 문법의 체인으로 좀 더 자연스러운 답변을 받을 수 있도록 조정 가능

```
from operator import itemgetter
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import RunnablePassthrough

answer_prompt = PromptTemplate.from_template(
    """Given the following user question, corresponding SQL query, and SQL result

Question: {question}
SQL Query: {query}
SQL Result: {result}
```

```
Answer: """
)

answer = answer_prompt | gemini_lc | StrOutputParser()
```

```
# 생성한 쿼리를 실행 → 결과를 출력하기 위한 체인 생성하기
chain_5 = (
    RunnablePassthrough.assign(query=write_query).assign(
        result=itemgetter("query") | execute_query
    )
    | answer
)
```

```
# 실행 결과 확인해보기

chain_5.invoke({"question": "테디의 transaction 의 합계를 구하세요"})
```

- `'테디의 transaction 합계는 -965.7입니다.'` - (`1.3s`)

---

3) `Agent`

- **`Agent` 활용 → `Sql` `쿼리` 생성 → `실행 결과`를 답변 → `출력` 가능**

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_community.utilities import SQLDatabase
from langchain_community.agent_toolkits import create_sql_agent

import os
from dotenv import load_dotenv

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,                                          # temperature = 0
    max_output_tokens=4096,
)
```

- `gemini-2.5-flash-lite`

```
E0000 00:00:1760244495.123340 4613744 alts_credentials.cc:93] ALTS creds ignore
```

```
# SQLite 데이터베이스에 연결하기
db_2 = SQLDatabase.from_uri("sqlite:///data/finance.db")
```

```
# Agent 생성
# agent_type 변경함
# 이 타입은 Gemini 모델의 도구 사용 능력(Function Calling)을 최적으로 활용함

agent_executor = create_sql_agent(
    llm=gemini_lc,
    db=db_2,
    agent_type="tool-calling",
    verbose=True                                    # 에이전트의 사고 과정을 출력함
)
```

```
# 실행 결과 확인

agent_executor.invoke(
    {"input": "테디와 셜리의 transaction 의 합계를 구하고 비교하세요"}
)
```

- **`Agent` 활용한 실행 결과 확인하기** - (`21.0s`)
  - 과정 - *`verbose=True`*

```
**Entering new SQL Agent Executor chain...**

*Invoking: `sql_db_list_tables` with `{}`*
*responded:  Finally, I will construct a query to sum the transactions for



**accounts, customers, transactions**
*Invoking: `sql_db_schema` with `{'table_names': 'customers, transactions'}`*



CREATE TABLE customers (
    customer_id INTEGER,
    name TEXT,
    age INTEGER,
    email TEXT,
    PRIMARY KEY (customer_id)
)

/*
3 rows from customers table:
customer_id    name    age    email
1    테디    30    teddy@example.com
2    폴    40    paul@example.com
```

```
3    셜리    25    shirley@example.com
*/


CREATE TABLE transactions (
    transaction_id INTEGER,
    account_id INTEGER,
    amount REAL,
    transaction_date TEXT,
    PRIMARY KEY (transaction_id),
    FOREIGN KEY(account_id) REFERENCES accounts (account_id)
)

/*
3 rows from transactions table:
transaction_id    account_id    amount    transaction_date
1    1    74.79    2024-07-13
2    1    -224.1    2024-05-13
3    1    -128.9    2024-01-25
*/

*Invoking: `sql_db_schema` with `{'table_names': 'accounts'}`*



CREATE TABLE accounts (
    account_id INTEGER,
    customer_id INTEGER,
    balance REAL,
    PRIMARY KEY (account_id),
    FOREIGN KEY(customer_id) REFERENCES customers (customer_id)
)

/*
3 rows from accounts table:
account_id    customer_id    balance
1    1    1000.5
2    2    2500.75
3    3    1500.0

*Invoking: `sql_db_query_checker` with `{'query': "SELECT SUM(T2.amount) FR
responded: I need to join the `customers`, `accounts`, and `transactions` t

First, I will get the sum of transactions for Teddy.

```sql
SELECT SUM(T2.amount) FROM customers AS T1 INNER JOIN accounts AS T3 ON T1.
```
```

Invoking: `sql_db_query` *with* `{'query': "SELECT SUM(T2.amount) FROM customers AS T1 INNER JOIN accounts AS T3 ON T1.customer_id = T3.customer_id INNER JOIN transactions AS T2 ON T3.account_id = T2.account_id WHERE T1.name = '테디'"}`

Invoking: `sql_db_query_checker` *with* `{'query': "SELECT SUM(T2.amount) FROM customers AS T1 INNER JOIN accounts AS T3 ON T1.customer_id = T3.customer_id INNER JOIN transactions AS T2 ON T3.account_id = T2.account_id WHERE T1.name = '셜리'"}`

SELECT SUM(T2.amount) FROM customers AS T1 INNER JOIN accounts AS T3 ON T1.customer_id = T3.customer_id INNER JOIN transactions AS T2 ON T3.account_id = T2.account_id WHERE T1.name = '셜리'

Invoking: `sql_db_query` *with* `{'query': "SELECT SUM(T2.amount) FROM customers AS T1 INNER JOIN accounts AS T3 ON T1.customer_id = T3.customer_id INNER JOIN transactions AS T2 ON T3.account_id = T2.account_id WHERE T1.name = '셜리'"}`

테디의 거래 합계는 -965.7 이고 셜리의 거래 합계는 656.64 입니다. 셜리의 거래 합계가 테디의 거래 합계보다 더 큽니다.

Finished chain.

- 결과

```
{'input': '테디와 셜리의 transaction 의 합계를 구하고 비교하세요',
'output': '테디의 거래 합계는 -965.7 이고 셜리의 거래 합계는 656.64 입니다. 셜리의 거래 합
```

- next: ***`03. 구조화된 출력 체인 (with_structered_output)`***
