


The background is a vibrant, abstract composition of light trails in shades of blue, teal, and yellow, creating a sense of motion and depth. A large, white, rounded rectangular box is centered in the middle of the image, serving as a backdrop for the text.

FastAPI その2

この資料について



オンライン学習プラットフォーム『Udemy』
で公開している、
『Python x FastAPI初心者向け講座』の
説明用資料です

[https://www.udemy.com/course/
python_fastapi](https://www.udemy.com/course/python_fastapi)

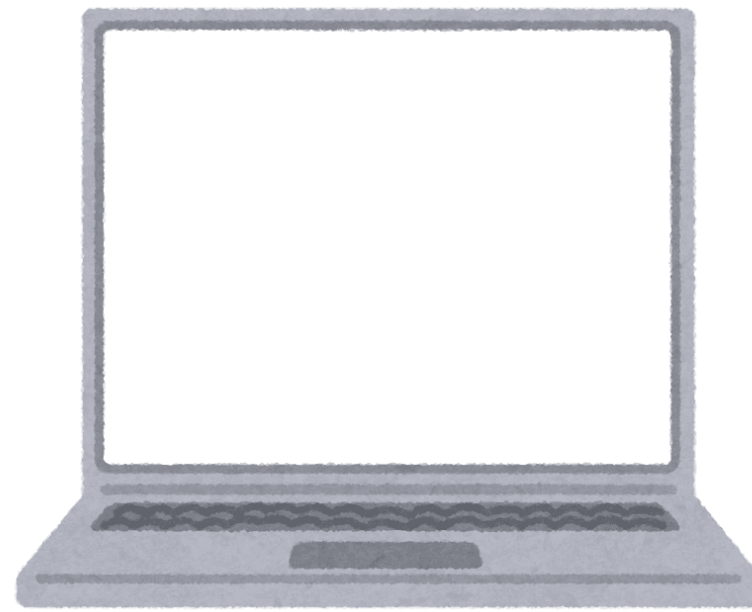


データベースと 接続

データベース

実運用では

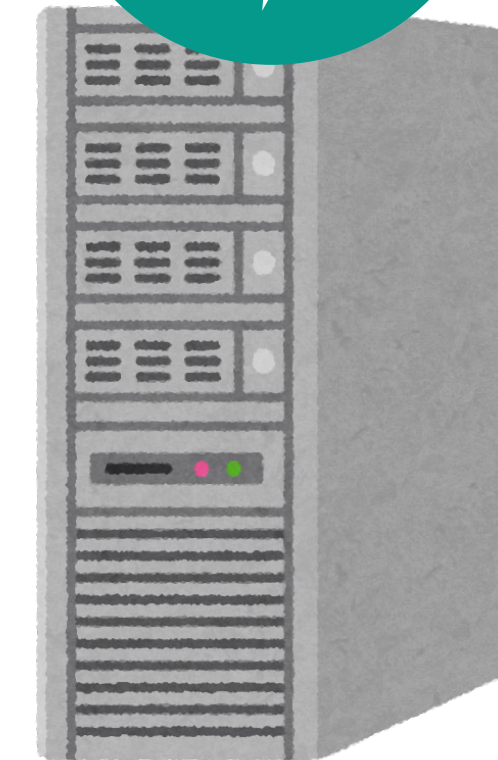
データベースにデータ保存される



リクエスト

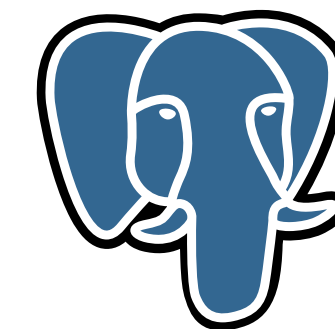


レスポンス



MySQL™

SQLite



データベース比較表

DB種類	リレーショナル データベース (RDB)	NoSQL	ベクトル データベース (VectorDB)
特徴	テーブル形式でデータを管理し、SQLを使用してデータを操作する。データ間の関係性を重視。	キーバリューストア、ドキュメントストア、ワイドカラムストア、グラフデータベースなど多様な形式がある。	データを高次元ベクトルとして格納し、類似性検索が得意。AIや機械学習モデルの結果を効率的に検索可能。
用途	トランザクションの処理、会計システム、在庫管理などデータの整合性が重要なアプリ。	大量のデータや構造が不定なデータを扱うアプリ、リアルタイムWebアプリ、ビッグデータ解析。	画像検索、推薦システム、自然言語処理での類似性検索、高度な分析アプリ。
向いていない事	大規模な非構造化データの管理 高い拡張性が要求されるシステム	厳密なトランザクション管理 複雑な結合が多用されるアプリ	トランザクション管理、 従来型のDB操作が主体のアプリ
サービス	MySQL, PostgreSQL, SQLite	MongoDB, Cassandra, Neo4j	Milvus, Faiss, Pinecone

ORM



Object/RDB マッピング

データベース操作にはSQLの知識が必要

データベースによってコマンドが若干変わる

データベース操作をプログラミングで

対応できるようにした仕組み

Pythonの場合 ・ SQLAlchemy, SQLModel etc...

SQLAlchemy



PythonのORMライブラリでは最も有名
情報豊富、SQLインジェクション対策済み
1.4以降 非同期対応が進んでいる

<https://www.sqlalchemy.org/>

インストール

```
$ pip install sqlalchemy==2.0.27
```

```
$ pip install greenlet==3.0.3 # 非同期対応ライブラリ
```

```
$ pip install aiosqlite==0.20.0 # SQLite非同期対応
```


事前知識



非同期処理と同期処理が混在すると思われぬエラーが発生

この講座では非同期対応を解説

エンジン・・・DBと接続する道具

セッション・・・DBとの通信経路

DBとの情報通信はセッションが使われる

(接続～切断までの一連の通信)

非同期対応の場合は `AsyncSession`

データベース接続関数

database.py

```
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
```

```
from sqlalchemy.orm import sessionmaker, declarative_base
```

```
DB_URL = "sqlite+aiosqlite:///fastapi-app.db" # SQLiteの非同期対応
```

```
engine = create_async_engine(DB_URL, echo=True)
```

```
Base = declarative_base()
```

```
# DBセッションオブジェクトを生成
```

```
db_session = sessionmaker(
```

```
    autocommit=False,
```

```
    autoflush=False,
```

```
    bind=engine,
```

```
    class_=AsyncSession)
```

```
async def get_db():
```

```
    async with db_session() as session:
```

```
        yield session
```


コードの解説

```
engine = create_async_engine(DB_URL, echo=True)
```

非同期エンジン生成 echo=True 実行されるSQL文を表示 (本番環境ではFalseにする)

```
Base = declarative_base() ・ ・ DBテーブルのベースクラス
```

```
db_session = sessionmaker(autocommit=False, autoflush=False, bind=engine,  
class_=AsyncSession)
```

sessionmaker ・ ・ DBセッションを作成

autocommit=False ・ ・ コミットするまでDBに反映されない

autoflush=False ・ ・ セッションの変更が即時実行されない

bind=engine ・ ・ DBセッションが扱うDBエンジン指定

class_=AsyncSession ・ ・ 扱うセッションクラスの指定

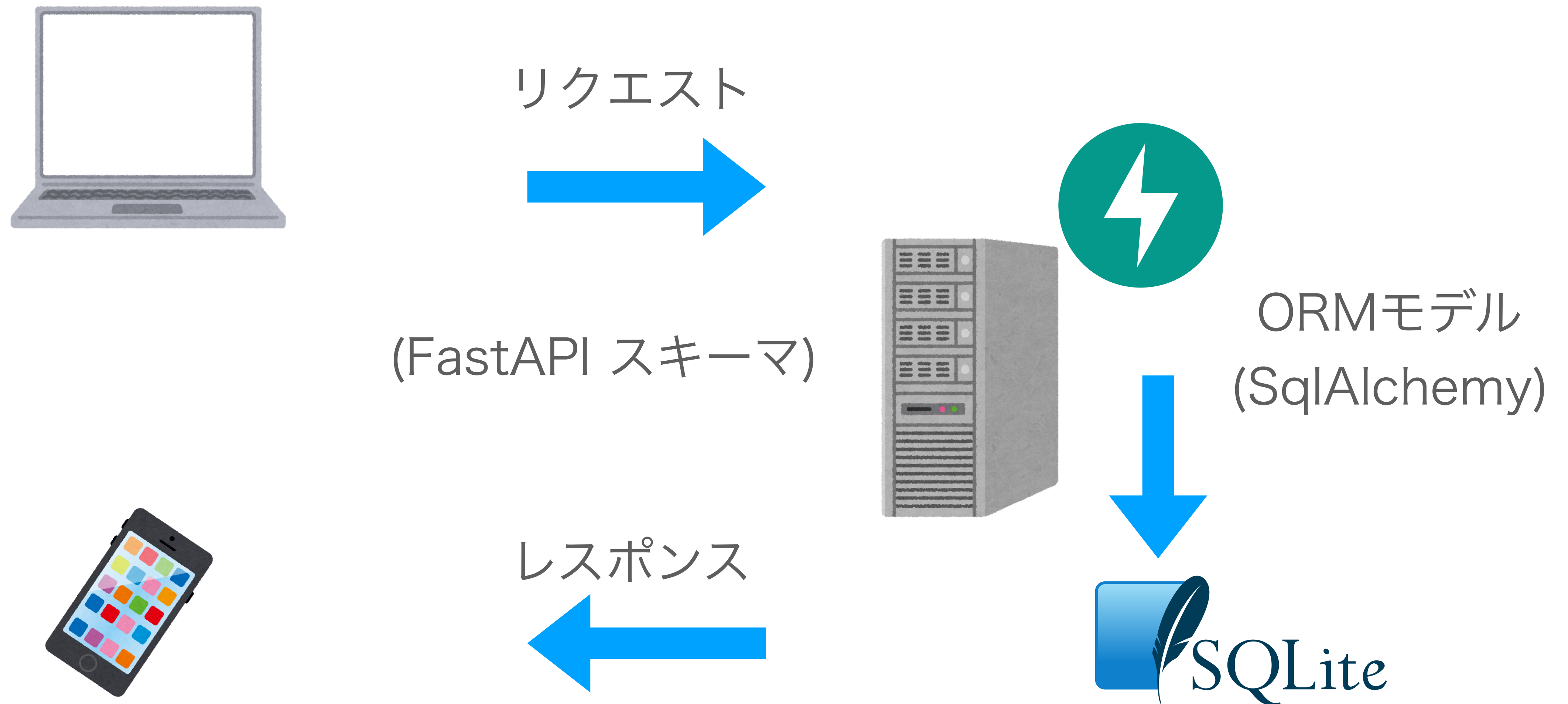
```
async with db_session as session:
```

```
    yield session
```

with ・ ・ DB接続などで使われる セッションの開始と終了を自動的に管理

yield session ・ ・ 呼び出し元にセッションを提供 (ジェネレータ)

スキーマとORMモデル



DB接続(ORM)のモデル作成

※Pydanticと文法が違うので注意

models/__init__.py 空ファイル

models/contact.py

```
from sqlalchemy import Column, Integer, String, Boolean, DateTime
```

```
from database import Base
```

```
from datetime import datetime
```

```
class Contact(Base): # 継承
```

```
    __tablename__ = "contacts" # テーブル名
```

```
    id = Column(Integer, primary_key=True, autoincrement=True) # 主キー、自動インクリメント
```

```
    name = Column(String(50), nullable=False)
```

```
    email = Column(String(255), nullable=False)
```

```
    url = Column(String(255), nullable=True)
```

```
    gender = Column(Integer, nullable=False) # enumで設定するパターンもある
```

```
    message = Column(String(200), nullable=False)
```

```
    is_enabled = Column(Boolean, default=False)
```

```
    created_at = Column(DateTime, default=datetime.now())
```

id	name	email
1	田中	
2	山田	
3	田中	
4	佐藤	

マイグレーション

作成したDB接続(ORM)モデルを元にDBテーブルを生成する

migrate_db.py

```
from sqlalchemy.ext.asyncio import create_async_engine
```

```
from models.contact import Base
```

```
import asyncio
```

```
DB_URL = "sqlite+aiosqlite:///fastapi-app.db"
```

```
engine = create_async_engine(DB_URL, echo=True)
```

```
async def reset_database():
```

```
    async with engine.begin() as conn: # with 例外発生時でもリソース解放される
```


```
        await conn.run_sync(Base.metadata.drop_all) # テーブル削除
```

```
        await conn.run_sync(Base.metadata.create_all) # テーブル作成
```

```
if __name__ == "__main__": # スクリプトで実行時のみ実行
```

```
    asyncio.run(reset_database())
```

データベース生成確認



```
$ python migrate_db.py
```

データベースと

その中のテーブルが生成された事を確認

```
$ sqlite3 fast-app.db
```

```
sqlite> .tables
```

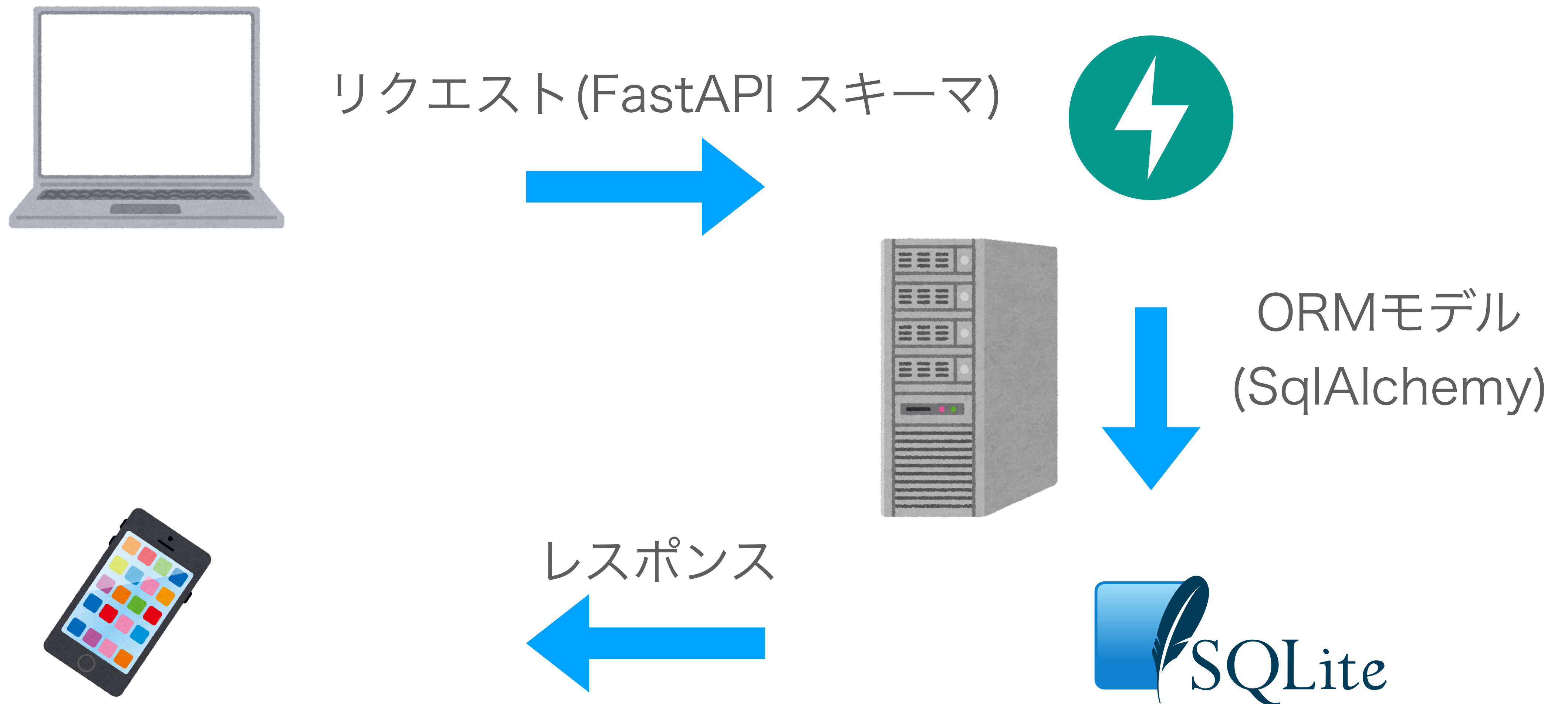
```
contacts
```

抜けるには Ctrl + D



スキーマモデル の修正

スキーマとORMモデル



情報の入出力



全件表示 ・ ・ id, name, created_at

保存 ・ ・ idとcreated_atは自動設定

詳細表示 ・ ・ 全情報

通信内容を絞るために

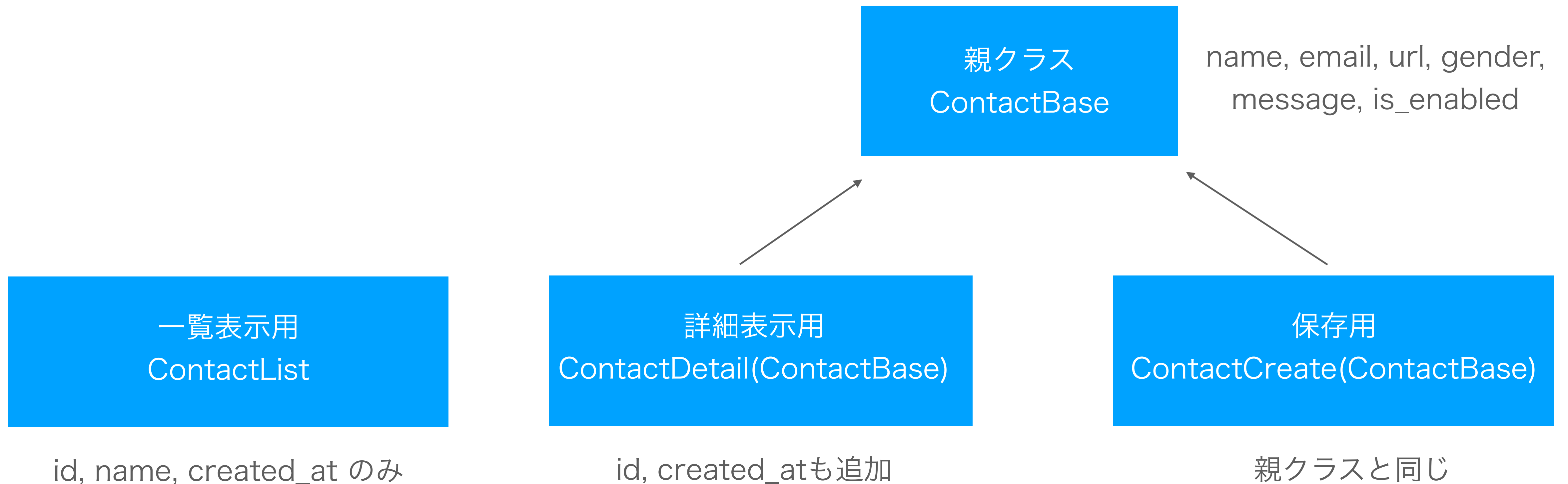
Pydanticモデルの定義を変更して対応

(sqlAlchemyのselectで絞ってもPydanticで自動補完されるため)

スキーマのモデルクラス

継承を使って分離する

(共通部分を親クラスに持たせる)



schemas/contact.py 1

```
from datetime import datetime
from pydantic import BaseModel, Field, EmailStr, HttpUrl
```

```
# 一覧表示用モデル
```

```
class ContactList(BaseModel):
```

```
    id: int
```

```
    name: str = Field(..., min_length=2, max_length=50)
```

```
    created_at: datetime
```

```
class Config: # ORMモデルと紐付け
```

```
    from_attributes = True
```

schemas/contact.py 2

ベースモデル

```
class ContactBase(BaseModel):
```

```
    name: str = Field(..., min_length=2, max_length=50)
```

```
    email: EmailStr
```

```
    url: HttpUrl | None = Field(default=None)
```

```
    gender: int = Field(..., strict=True, ge=0, le=2)
```

```
    message: str = Field(..., max_length=200)
```

```
    is_enabled: bool = Field(default=False)
```

```
class Config:
```

```
    from_attributes = True
```


schemas/contact.py 3

詳細表示用モデル

```
class ContactDetail(ContactBase):
```

```
    id: int
```

```
    created_at: datetime
```

```
class Config:
```

```
    from_attributes = True
```

保存用モデル


```
class ContactCreate(ContactBase):
```

```
    pass
```




CRUD (Create)

ファットコントローラを防ぐ



処理がルータに集まってしまうと

ファイル行数が増える

(読みづらい、メンテしづらい)

処理部分はコントローラ(MVCモデルの場合)

Fat Controller防止のため処理を分ける

cruds/contact.py

```
from sqlalchemy.ext.asyncio import AsyncSession
```

```
import models.contact as contact_model # ORMモデル
```

```
import schemas.contact as contact_schema # スキーマモデル
```

```
async def create_contact(db: AsyncSession, contact: contact_schema.ContactCreate) ->
```

```
contact_model.Contact:
```

```
    """
```

DBに新規保存

引数:

db: DBセッション

contact: 作成するコンタクトのデータ

戻り値:

作成されたORMモデル

```
    """
```

```
contact_data = contact.model_dump()
```

```
if contact_data["url"] is not None: # urlフィールドが存在し、Noneでない場合は文字列に変換
```

```
    contact_data["url"] = str(contact_data["url"])
```


cruds/contact.py

```
db_contact = contact_model.Contact(**contact_data) # db保存  
はsqlalchemyのモデル
```

```
db.add(db_contact) # 追加
```

```
await db.commit() # コミット (反映
```

```
await db.refresh(db_contact) # リフレッシュ
```

```
return db_contact # ORMモデルを返す
```

ルーターも編集

`routers/contact.py`

```
from fastapi import APIRouter, Depends # 依存性注入
from sqlalchemy.ext.asyncio import AsyncSession # DBセッション
import schemas.contact as contact_schema # スキーマモデル
import cruds.contact as contact_crud # DB保存関数
from database import get_db # DBセッション取得関数

略
```

```
# 保存
```

```
@router.post("/contacts", response_model=contact_schema.ContactCreate)
async def create_contact(body: contact_schema.ContactCreate, db:
AsyncSession = Depends(get_db)): # 第二引数にセッションを渡してる
    return await contact_crud.create_contact(db, body)
```


ブラウザで確認

POST /contacts Create Contact ^

Parameters Cancel

No parameters

Request body required application/json v

```
{
  "name": "string",
  "email": "user@example.com",
  "url": "https://example.com/",
  "gender": 0,
  "message": "string",
  "is_enabled": false
}
```

Execute

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/contacts' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "string",
    "email": "user@example.com",
    "url": "https://example.com/",
    "gender": 0,
    "message": "string",
    "is_enabled": false
  }'
```

Request URL

http://localhost:8000/contacts

Server response

Code	Details
200	<div>Response body</div> <div><pre>{ "name": "string", "email": "user@example.com", "url": "https://example.com/", "gender": 0, "message": "string", "is_enabled": false }</pre></div> <div>Download</div>

sqlite側も確認

別ターミナルで確認

(登録毎にidが1ずつ上がっている事を確認)

```
$ sqlite3 fast-app.db
```

```
sqlite> select * from contacts;
```

```
1|string|user@example.com|https://example.com/|
```

```
0|string|0|2024-03-04 11:40:05.104390
```

```
2|string|user@example.com|https://example.com/|
```

```
0|string|0|2024-03-04 11:41:12.570900
```




DI (Depends)

依存性注入

Dependency Injection(依存性注入) 略して DI

関数、クラス、インスタンス、インスタンスメソッドなどに適用できる

db: 型ヒント

```
db: AsyncSession = Depends(get_db))
```

get_db()はセッション取得関数

セッション取得を別の関数に任せる事で

ルーティングに専念できる

(責任の分離・再利用)

DIのサンプル

`routers/contact.py`

依存性注入のサンプル

依存関数：特定のメッセージを返す

```
def get_message():
```

```
    message = "Hello, World!"
```

```
    print(f"get_messageが実行されたよ: {message}")
```

```
    return message
```

APIエンドポイント：依存関数からメッセージを受け取って表示

```
@router.get("/depends")
```

```
async def main(message: str = Depends(get_message)):
```

```
    print(f"エンドポイントにアクセスがあったよ: {message}")
```

```
    return {"message": message}
```




CRUD (Read)

CRUD Read GET通信

`cruds/contact.py` 略

```
from typing import List, Tuple
```

```
from sqlalchemy import select
```

```
from sqlalchemy.engine import Result
```

```
from datetime import datetime
```

```
# 一覧表示 # sqlalchemyを使ってDBから情報取得
```

```
async def get_contact_all(db: AsyncSession) -> List[Tuple[int, str, datetime]]:
```

```
    result : Result = await db.execute(
```

```
        select(
```

```
            contact_model.Contact.id,
```

```
            contact_model.Contact.name,
```

```
            contact_model.Contact.created_at)
```

```
    )
```

```
    return result.all() # 全て実行
```

CRUD Read GET通信 ルーター

router/contact.py

一覧表示

複数表示するのでlist[] で配列として指定

@router.get("/contacts",

response_model=list[contact_schema.ContactList])

引数にdb接続(セッション)

async def get_contact_all(db: AsyncSession = Depends(get_db)):

Read処理を実行

return await contact_crud.get_contact_all(db)

もう一つのRead (詳細表示)

cruds/contact.py

略

詳細表示

idを指定して条件を取得。where句でidを指定

idが存在しない可能性もあるのでNoneも返り値の型に設定

```
async def get_contact(db: AsyncSession, id: int) -> contact_model.Contact | None:
```

```
    query = select(contact_model.Contact).where(contact_model.Contact.id == id)
```

```
    result : Result = await db.execute(query)
```

```
    return result.scalars().first() # scalars() 単一の値を取得、first() 最初の要素を取得
```

もう一つのRead ルーター

`routers/contact.py`

```
import fastapi from HTTPException
```

```
～ 略 ～
```

```
# 詳細表示
```

```
# idが存在しなかったら404を返すように例外処理も対応
```

```
@router.get("/contacts/{id}", response_model=contact_schema.ContactDetail)
```

```
async def get_contact(id: int, db: AsyncSession = Depends(get_db)):
```

```
    contact = await contact_crud.get_contact(db, id) # 関数実行
```

```
    if contact is None: # もし Noneだったら例外発生
```

```
        raise HTTPException(status_code=404, detail="Contact not found")
```

```
    return contact
```




CRUD (Update)

CRUD更新処理

`cruds/contact.py`

`# 更新`

```
async def update_contact(
    db: AsyncSession,
    contact: contact_schema.ContactCreate, # 更新したい情報
    original: contact_model.Contact # DB保存済みの情報
) -> contact_model.Contact:
    original.name = contact.name
    original.email = contact.email
    if original.url is not None: # urlは文字列に変換
        original.url = str(contact.url)
    original.gender = contact.gender
    original.message = contact.message
    db.add(original)
    await db.commit()
    await db.refresh(original)
    return original
```


更新処理 ルーター

`routers/contact.py`

`# 更新`

`@router.put("/contacts/{id}", response_model=contact_schema.ContactCreate)`

`async def update_contact(id: int, body: contact_schema.ContactCreate, db:`

`AsyncSession = Depends(get_db)):`

`# 存在するかどうかのチェック`

`contact = await contact_crud.get_contact(db, id)`

`if contact is None:`

`raise HTTPException(status_code=404, detail="Contact not found")`

`return await contact_crud.update_contact(db, body, original=contact)`



CRUD (Delete)

CRUD Delete 削除処理



cruds/contact.py

削除

```
async def delete_contact(db: AsyncSession,  
original: contact_model.Contact) -> None:  
    await db.delete(original)  
    await db.commit()
```

Delete ルーター



削除

```
@router.delete("/contacts/{id}", response_model=None)
```

```
async def delete_contact(id: int, db: AsyncSession = Depends(get_db)):
```

```
    # 存在の確認
```

```
    contact = await contact_crud.get_contact(db, id)
```

```
    if contact is None:
```

```
        raise HTTPException(status_code=404, detail="Contact not found")
```

```
    # 存在していたら実行
```

```
    return await contact_crud.delete_contact(db, original=contact)
```