# フェーズ1実施報告および解説書

---

## 2. 実施内容の詳細

### 2.1 作成したファイル一覧

| ファイル名 | 役割 |

|-----------|------|

| `backend/requirements.txt` | Pythonパッケージの依存関係定義 |

| `backend/.env` | データベース接続情報などの環境変数 |

| `backend/database.py` | データベース接続設定とセッション管理 |

| `backend/models.py` | SQLAlchemyのORMモデル定義 |

| `backend/schemas.py` | PydanticによるAPIのリクエスト/レスポンス型定義 |

| `backend/main.py` | FastAPIアプリケーション本体 |

### 2.2 ディレクトリ構造

```

backend/

├── .env                          # 環境変数（DB接続情報）

├── requirements.txt              # Pythonパッケージ依存関係

├── database.py                   # DB接続設定

├── models.py                     # SQLAlchemyモデル

├── schemas.py                    # Pydanticスキーマ

├── main.py                       # FastAPIアプリ本体

└── create\_sim\_result\_table.sql   # DBテーブル作成SQL（既存）

```

---

## 3. 技術解説

### 3.1 requirements.txt - パッケージ依存関係

```txt

fastapi==0.115.0          # WebフレームワークFastAPI本体

uvicorn[standard]==0.31.0 # ASGIサーバー（FastAPIを起動するためのサーバー）

sqlalchemy==2.0.35        # PythonのORM（データベース操作ライブラリ）

pymysql==1.1.1            # MySQLドライバ

cryptography==43.0.1      # PyMySQLの暗号化機能に必要

python-dotenv==1.0.1      # .envファイルから環境変数を読み込むライブラリ

websockets==13.1          # WebSocket通信用（フェーズ2で使用予定）

```

**\*\*学習ポイント\*\***:

- **\*\*FastAPI\*\***: Pythonの高速Webフレームワーク。型ヒントを活用した自動バリデーションとドキュメント生成が特徴

- **\*\*Uvicorn\*\***: ASGIサーバー。FastAPIアプリを実際にHTTPサーバーとして起動する役割

- **\*\*SQLAlchemy\*\***: Pythonで最も使われているORM。PythonのクラスをDBのテーブルにマッピング

### 3.2 database.py - データベース接続設定

**\*\*役割\*\***: MySQLデータベースへの接続を管理し、SQLAlchemyのセッションを提供

```python

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

import os

from dotenv import load\_dotenv

# .envファイルから環境変数を読み込む

load\_dotenv()

# 環境変数からDB接続情報を取得

DB\_HOST = os.getenv("DB\_HOST", "127.0.0.1")

DB\_PORT = os.getenv("DB\_PORT", "3306")

DB\_USER = os.getenv("DB\_USER", "root")

DB\_PASSWORD = os.getenv("DB\_PASSWORD", "")

DB\_NAME = os.getenv("DB\_NAME", "flexsim\_demo")

# MySQLへの接続URL作成

DATABASE\_URL = f"mysql+pymysql://{DB\_USER}:{DB\_PASSWORD}@{DB\_HOST}:{DB\_PORT}/{DB\_NAME}"

# データベースエンジン作成（echo=Trueで実行されるSQLをコンソールに表示）

engine = create\_engine(DATABASE\_URL, echo=True)

# セッションファクトリ作成

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# モデルの基底クラス

Base = declarative\_base()

# 依存性注入用のDB取得関数

def get\_db():

    db = SessionLocal()

    try:

        yield db  # FastAPIのDependsで使用

    finally:

        db.close()  # 必ずセッションを閉じる

```

**\*\*学習ポイント\*\***:

- **\*\*create\_engine\*\***: SQLAlchemyがDBに接続するためのエンジンオブジェクトを作成

- **\*\*sessionmaker\*\***: DBとのやり取りを行う「セッション」を作成するファクトリ

- **\*\*declarative\_base\*\***: SQLAlchemyのモデルクラスの基底クラス

- **\*\*get\_db関数\*\***: FastAPIの依存性注入機能で使用。各リクエストごとに新しいセッションを作成し、処理後に必ず閉じる

### 3.3 models.py - SQLAlchemyモデル定義

**\*\*役割\*\***: データベースのテーブルをPythonのクラスとして表現（ORM）

```python

from sqlalchemy import Column, Integer, String, Text, DECIMAL, ForeignKey

from sqlalchemy.orm import relationship

from database import Base

class Scenario(Base):

    \_\_tablename\_\_ = "scenarios"  # テーブル名を指定

    # カラム定義

    scenario\_id = Column(Integer, primary\_key=True, autoincrement=True)

    scenario\_name = Column(String(255), nullable=False)

    description = Column(Text, nullable=True)

    # リレーションシップ（1対多）

    results = relationship("SimulationResult", back\_populates="scenario")

class SimulationResult(Base):

    \_\_tablename\_\_ = "simulation\_results"

    sim\_result\_id = Column(Integer, primary\_key=True, autoincrement=True)

    scenario\_id = Column(Integer, ForeignKey("scenarios.scenario\_id"), nullable=False)

    replication = Column(Integer, nullable=False)

    total\_labor\_costs = Column(DECIMAL(10, 2), nullable=False)

    ontime\_delivery\_rate = Column(DECIMAL(5, 4), nullable=False)

    # リレーションシップ（多対1）

    scenario = relationship("Scenario", back\_populates="results")

```

**\*\*学習ポイント\*\***:

- **\*\*Column\*\***: テーブルのカラムを定義。型、制約、デフォルト値などを指定可能

- **\*\*relationship\*\***: テーブル間のリレーションシップを定義。`Scenario.results`で関連する結果データにアクセス可能

- **\*\*ForeignKey\*\***: 外部キー制約を定義。参照整合性を保証

**\*\*ORMの利点\*\***:

```python

# SQLを書かずにPythonでデータベース操作ができる

scenario = db.query(Scenario).filter(Scenario.scenario\_id == 1).first()

print(scenario.scenario\_name)  # シナリオ名を取得

print(len(scenario.results))   # 関連する結果データの件数

```

### 3.4 schemas.py - Pydanticスキーマ定義

**\*\*役割\*\***: APIのリクエスト/レスポンスのデータ型を定義（バリデーションと型安全性）

```python

from pydantic import BaseModel

from typing import Optional

from decimal import Decimal

# ベースクラス（共通フィールド）

class ScenarioBase(BaseModel):

    scenario\_name: str

    description: Optional[str] = None

# 作成時のスキーマ（IDは不要）

class ScenarioCreate(ScenarioBase):

    pass

# レスポンス用スキーマ（IDを含む）

class Scenario(ScenarioBase):

    scenario\_id: int

    class Config:

        from\_attributes = True  # ORMモデルから変換可能にする

# ... (SimulationResult系も同様)

# 集計結果のレスポンス用スキーマ

class ScenarioSummary(BaseModel):

    scenario\_id: int

    scenario\_name: str

    average\_total\_costs: Decimal

    average\_delivery\_rate: Decimal

```

**\*\*学習ポイント\*\***:

- **\*\*Pydantic\*\***: Pythonのデータ検証ライブラリ。型ヒントを使って自動的にバリデーションを行う

- **\*\*BaseModel\*\***: Pydanticのモデルの基底クラス

- **\*\*from\_attributes = True\*\***: SQLAlchemyのORMモデルから自動的にPydanticモデルに変換できるようにする設定

**\*\*SQLAlchemyモデル vs Pydanticモデル\*\***:

- **\*\*SQLAlchemyモデル（models.py）\*\***: データベースのテーブル構造を表現

- **\*\*Pydanticモデル（schemas.py）\*\***: APIのリクエスト/レスポンスの形式を表現

### 3.5 main.py - FastAPIアプリケーション本体

**\*\*役割\*\***: FastAPIアプリケーションを定義し、APIエンドポイントを実装

```python

from fastapi import FastAPI, Depends

from fastapi.middleware.cors import CORSMiddleware

from sqlalchemy.orm import Session

from sqlalchemy import func

from typing import List

from database import get\_db

import models

import schemas

# FastAPIアプリケーション作成

app = FastAPI(title="Simulation Demo API", version="1.0.0")

# CORS設定（フロントエンドからのアクセスを許可）

app.add\_middleware(

    CORSMiddleware,

    allow\_origins=["http://localhost:3000"],  # Next.jsのデフォルトポート

    allow\_credentials=True,

    allow\_methods=["\*"],  # すべてのHTTPメソッドを許可

    allow\_headers=["\*"],  # すべてのヘッダーを許可

)

# ルートエンドポイント

@app.get("/")

def read\_root():

    return {"message": "Simulation Demo API is running"}

# 結果取得API（指示書の仕様: GET /api/results）

@app.get("/api/results", response\_model=List[schemas.ScenarioSummary])

def get\_results(db: Session = Depends(get\_db)):

    # SQLAlchemyでJOINとGROUP BYを使って集計

    results = (

        db.query(

            models.Scenario.scenario\_id,

            models.Scenario.scenario\_name,

            func.avg(models.SimulationResult.total\_labor\_costs).label("average\_total\_costs"),

            func.avg(models.SimulationResult.ontime\_delivery\_rate).label("average\_delivery\_rate"),

        )

        .join(models.SimulationResult, models.Scenario.scenario\_id == models.SimulationResult.scenario\_id)

        .group\_by(models.Scenario.scenario\_id, models.Scenario.scenario\_name)

        .all()

    )

    # Pydanticモデルに変換してレスポンス

    return [

        schemas.ScenarioSummary(

            scenario\_id=row.scenario\_id,

            scenario\_name=row.scenario\_name,

            average\_total\_costs=row.average\_total\_costs,

            average\_delivery\_rate=row.average\_delivery\_rate,

        )

        for row in results

    ]

# シナリオ一覧取得API

@app.get("/api/scenarios", response\_model=List[schemas.Scenario])

def get\_scenarios(db: Session = Depends(get\_db)):

    scenarios = db.query(models.Scenario).all()

    return scenarios

```

**\*\*学習ポイント\*\***:

#### 1. **\*\*FastAPIのデコレータ\*\***

```python

@app.get("/api/results", response\_model=List[schemas.ScenarioSummary])

```

- `@app.get()`: GETリクエストのエンドポイントを定義

- `response\_model`: レスポンスの型を指定。自動的にバリデーションとドキュメント生成

#### 2. **\*\*依存性注入（Dependency Injection）\*\***

```python

def get\_results(db: Session = Depends(get\_db)):

```

- `Depends(get\_db)`: FastAPIが自動的に`get\_db()`関数を呼び出してDBセッションを注入

- 各リクエストごとに新しいセッションを作成し、処理後に自動的にクローズ

#### 3. **\*\*CORS（Cross-Origin Resource Sharing）\*\***

- フロントエンド（http://localhost:3000）からバックエンド（http://localhost:8000）へのアクセスを許可

- セキュリティ上、同じオリジン以外からのアクセスはデフォルトでブロックされるため、明示的に許可が必要

#### 4. **\*\*SQLAlchemyのクエリ\*\***

```python

db.query(...)

  .join(...)

  .group\_by(...)

  .all()

```

- SQLを書かずに、Pythonのメソッドチェーンでクエリを構築

- `func.avg()`: SQLの集計関数AVGを使用

---

## 4. API仕様

### 4.1 GET / - ヘルスチェック

**\*\*エンドポイント\*\***: `GET http://localhost:8000/`

**\*\*レスポンス\*\***:

```json

{

  "message": "Simulation Demo API is running"

}

```

### 4.2 GET /api/results - シミュレーション結果取得

**\*\*エンドポイント\*\***: `GET http://localhost:8000/api/results`

**\*\*説明\*\***: 各シナリオの平均値を集計して返す

**\*\*レスポンス例\*\***:

```json

[

  {

    "scenario\_id": 1,

    "scenario\_name": "現状シナリオ",

    "average\_total\_costs": 9249.12,

    "average\_delivery\_rate": 0.85

  },

  {

    "scenario\_id": 2,

    "scenario\_name": "人員1名追加",

    "average\_total\_costs": 10500.00,

    "average\_delivery\_rate": 0.92

  }

]

```

### 4.3 GET /api/scenarios - シナリオ一覧取得

**\*\*エンドポイント\*\***: `GET http://localhost:8000/api/scenarios`

**\*\*説明\*\***: 登録されているシナリオ一覧を返す

**\*\*レスポンス例\*\***:

```json

[

  {

    "scenario\_id": 1,

    "scenario\_name": "現状シナリオ",

    "description": "現在の人員配置でのシミュレーション"

  }

]

```

---

## 5. 動作確認手順

### 5.1 環境構築

1. **\*\*Pythonパッケージのインストール\*\***

```bash

cd backend

pip install -r requirements.txt

```

2. **\*\*環境変数の設定\*\***

`backend/.env`ファイルのDB\_PASSWORDを設定（必要に応じて）

3. **\*\*データベースの確認\*\***

MySQL Workbench等で以下を確認：

- `flexsim\_demo`スキーマが存在すること

- `scenarios`テーブルと`simulation\_results`テーブルが存在すること

### 5.2 サーバー起動

```bash

cd backend

uvicorn main:app --reload --host 0.0.0.0 --port 8000

```

**\*\*オプション説明\*\***:

- `main:app`: `main.py`の`app`オブジェクトを起動

- `--reload`: コード変更時に自動再起動（開発時に便利）

- `--host 0.0.0.0`: すべてのネットワークインターフェースでリッスン

- `--port 8000`: ポート8000で起動

### 5.3 動作確認

#### 1. ブラウザでアクセス

- **\*\*API\*\***: http://localhost:8000/

- **\*\*自動生成ドキュメント\*\***: http://localhost:8000/docs

#### 2. APIドキュメントで動作確認

1. http://localhost:8000/docs にアクセス

2. `GET /api/scenarios` をクリック

3. 「Try it out」→「Execute」をクリック

4. レスポンスを確認

#### 3. テストデータ投入（必要に応じて）

**\*\*シナリオ追加\*\***:

```sql

INSERT INTO scenarios (scenario\_name, description) VALUES

('現状シナリオ', '現在の人員配置'),

('人員1名追加', '作業員を1名追加したシナリオ'),

('残業1時間追加', '残業時間を1時間追加したシナリオ');

```

**\*\*シミュレーション結果追加\*\***:

```sql

INSERT INTO simulation\_results (scenario\_id, replication, total\_labor\_costs, ontime\_delivery\_rate) VALUES

(1, 1, 9000.00, 0.85),

(1, 2, 9500.00, 0.87),

(1, 3, 9250.00, 0.86);

```

---

## 6. フェーズ1の達成内容

✅ **\*\*完了項目\*\***:

1. FastAPIのプロジェクト構造作成

2. データベース接続設定（SQLAlchemy）

3. ORMモデル定義（Scenario, SimulationResult）

4. Pydanticスキーマ定義

5. 基本的なCRUD API実装（結果取得エンドポイント）

6. CORS設定（フロントエンドとの連携準備）

---

## 7. 次フェーズへの準備

フェーズ2では以下を実装予定：

1. **\*\*WebSocket機能\*\***: リアルタイム通知

2. **\*\*シミュレーション開始API\*\***: `POST /api/simulations`

3. **\*\*完了通知API\*\***: `POST /api/notify-completion`

4. **\*\*バックグラウンドタスク管理\*\***: Pythonプロセスの起動

---

## 8. 学習課題（推奨）

フェーズ2に進む前に、以下のキーワードについて調査いただくことをお勧めします：

### 8.1 WebSocket

- **\*\*キーワード\*\***: WebSocket, ASGI, FastAPI WebSocket

- **\*\*調査ポイント\*\***:

  - WebSocketとHTTPの違い

  - FastAPIでのWebSocket実装方法

  - コネクションマネージャーの実装パターン

### 8.2 バックグラウンドタスク

- **\*\*キーワード\*\***: FastAPI BackgroundTasks, asyncio, subprocess

- **\*\*調査ポイント\*\***:

  - FastAPIでのバックグラウンド処理

  - Pythonでの外部プロセス起動方法

  - 非同期処理の基礎

### 8.3 フロントエンド連携

- **\*\*キーワード\*\***: fetch API, WebSocket client, React hooks

- **\*\*調査ポイント\*\***:

  - ブラウザからのWebSocket接続方法

  - useEffectフックでのWebSocket管理

  - エラーハンドリングと再接続

---

## 9. 補足資料

### 9.1 FastAPI自動生成ドキュメントの使い方

FastAPIは、コードから自動的にAPIドキュメントを生成します：

- **\*\*Swagger UI\*\***: http://localhost:8000/docs

- **\*\*ReDoc\*\***: http://localhost:8000/redoc

ドキュメント上で直接APIをテストできるため、開発時に非常に便利です。

### 9.2 SQLAlchemyのクエリパターン

```python

# 基本的な検索

db.query(Model).all()  # すべて取得

db.query(Model).first()  # 最初の1件

db.query(Model).filter(Model.id == 1).all()  # 条件付き

# JOIN

db.query(Model1).join(Model2).all()

# 集計

db.query(func.count(Model.id)).scalar()

db.query(func.avg(Model.value)).scalar()

# 追加

new\_item = Model(name="test")

db.add(new\_item)

db.commit()

db.refresh(new\_item)  # DBから最新データを取得

# 更新

item = db.query(Model).first()

item.name = "updated"

db.commit()

# 削除

db.query(Model).filter(Model.id == 1).delete()

db.commit()

```

---

## 10. トラブルシューティング

### 10.1 データベース接続エラー

**\*\*エラー\*\***: `(pymysql.err.OperationalError) (2003, "Can't connect to MySQL server")`

**\*\*対処法\*\***:

1. MySQLサーバーが起動しているか確認

2. `.env`ファイルの接続情報が正しいか確認

3. ポート番号が3306であることを確認

### 10.2 モジュールインポートエラー

**\*\*エラー\*\***: `ModuleNotFoundError: No module named 'xxx'`

**\*\*対処法\*\***:

```bash

pip install -r requirements.txt

```

### 10.3 CORSエラー

**\*\*エラー\*\***: ブラウザコンソールに`CORS policy`のエラー

**\*\*対処法\*\***:

- `main.py`のCORSミドルウェア設定を確認

- フロントエンドのURLが`allow\_origins`に含まれているか確認

---

## 11. まとめ

フェーズ1では、FastAPIを使用したバックエンドの基盤を構築しました。

**\*\*主要な成果物\*\***:

- データベース接続設定

- ORMモデル定義

- API実装（結果取得エンドポイント）

**\*\*次のステップ\*\***:

- フェーズ2: WebSocket実装とシミュレーション制御機能の追加

ご質問があれば、お気軽にお声がけください。