JBS 中級コース(案1)／Day2／2026-05-21

Day1の設計書から FastAPIアプリを実装してテストする

Day1で起こした要件定義書・ER図・API仕様・シーケンス図をもとに、SQLAlchemy 2.0で実装し、ユニット結合の二層でカバレッジ80%を目指す3時間。

13:00〜16:00(オンライン)　FastAPI + SQLAlchemy 2.0 async　pytest + pytest-cov

本日のアジェンダ

13:00〜13:10　Session 01 Day1復習と本日のゴール
設計書の引き継ぎ確認と実装方針の合意。[10分]

13:10〜13:25　Session 02 プロジェクト構造の生成
routers／services／repositories の骨格を Claude Code で起こす。[15分]

13:25〜14:00　Session 03 データモデルの実装
SQLAlchemy 2.0 の Mapped 構文で全エンティティを実装、初期マイグレーション。[35分]

14:00〜14:15　Session 04 ユニットテストの追加
repository 層のユニットテストでカバレッジの基礎を作る。[15分]

14:25〜15:00　Session 05 API層の実装
router→service→repository の流れでチケット・コメント・SLAを積む。[35分]

15:00〜15:30 **Session 06 結合テストとカバレッジ80%**

TestClient で API を叩く結合テストを書き、CI を緑のまま保つ。[30分]

15:30〜15:50 **Session 07 動作確認とコードレビュー**

Swagger UI で挙動を確認、Strict Reviewer で点検。[20分]

15:50〜16:00 **まとめとDay3への引き継ぎ**

持ち帰る3点と Day3 (Docker・CI/CD) への宿題。[10分]

SESSION 01 ／ 13:00〜13:10

Day1復習と本日のゴール

Day1で作った設計書を引き継ぎ、Day2で何を実装するかを再確認します。

Day1成果物の引き継ぎ確認

- [x] `docs/requirements.md` 要件定義書、MoSCoW で機能分類
- [x] `docs/data_model.md` ER図、状態遷移図
- [x] `docs/openapi.yaml` API仕様、エラーコード一覧
- [x] `docs/sequence_diagrams.md` 主要3ユースケースのシーケンス図

> **Day1を欠席した方、設計書がない方**
> 講師から共有する完成版の設計書を `docs/` 配下に配置してから始めてください。Day2で実装する内容は設計書を読まないと判断できません。

Day2の3つのゴール

- [x] FastAPI + SQLAlchemy 2.0 async でチケット・コメント・SLA の3エンティティを動かす
- [x] pytest のユニットテストと結合テストでカバレッジ80%以上を確保する
- [x] CI を緑のまま保ちながらコミットを積み重ね、Day3の Docker 化に渡せる状態を作る

本日の進め方

> **レイヤを意識する**
> routers → services → repositories → models の単方向依存を守ってください。router にビジネスロジックを書くと、Day3でテストが書けません。`.github/copilot-instructions.md` にもこの方針が書かれているので、Claude Code に常に意識させます。

> **テストを後回しにしない**
> 「実装が終わってからテストを書く」では Day3の CI で破綻します。1機能実装したら1テスト書く、をリズムで回してください。

SESSION 02 ／ 13:10〜13:25

プロジェクト構造の生成

15分で routers／services／repositories／schemas の骨格を Claude Code に生成させます。

HANDS-ON

レイヤ別ディレクトリ構造を起こす

15分

1 環境確認とCI緑の維持

VSCode で配布リポジトリを開き、ターミナルで `source .venv/bin/activate` を実行。
続いて下記4つが全て成功することを確認します。

TERMINAL

```
ruff check app/ tests/ api/
ruff format --check app/ tests/ api/
mypy app/
pytest tests/ -v
```

2 Claude Code に骨格を起こさせる

Sidebar Chat または `claude` CLI で次のように依頼します。

PROMPT

```
docs/data_model.md と docs/openapi.yaml と CLAUDE.md を読み、
app/ 配下に下記のディレクトリ構造とスケルトンファイルを作成してください。

app/
  routers/
    tickets.py      /tickets エンドポイント (空関数のみ)
    comments.py     /tickets/{id}/comments エンドポイント
    sla.py          /sla/violations エンドポイント
    users.py        /users
    categories.py   /categories
  services/
    ticket_service.py
    comment_service.py
    sla_service.py
  repositories/
    ticket_repository.py
    comment_repository.py
    base.py             汎用 CRUD 基底
  models/
    __init__.py
    base.py             DeclarativeBase
    user.py
    category.py
    ticket.py
    comment.py
    sla_policy.py
    audit_log.py
  schemas/
    ticket.py
    comment.py
    user.py
    category.py
    sla.py
    errors.py           エラーレスポンス共通スキーマ
  exceptions.py         ドメイン例外
  dependencies.py       Depends 用のヘルパ

全ての関数に型ヒントを必ず付けてください。
import 順は isort 規約に従ってください。
ruff check と mypy が通る状態で出力してください。
```

3 生成されたコードのレビュー

Claude Code が出した差分を Apply する前に、各ファイルの import 文と関数シグネチャだけ目視で確認してください。`Mapped[T]` と `mapped_column` を使うSQLAlchemy 2.0構文になっているか、特に注意。

---

4 CIをグリーンに保ったまま commit

TERMINAL

```
ruff check app/ --fix
ruff format app/
mypy app/
pytest tests/
git add app/
git commit -m "feat: project skeleton for Day2"
git push
```

Actions タブで全ジョブが緑になることを確認してから次に進みます。

SESSION 03 ／ 13:25〜14:00

データモデルの実装

35分でSQLAlchemy 2.0のモデルを全エンティティ分実装し、初期マイグレーション相当を整備します。

HANDS-ON **User / Category / Ticket / Comment / SLAPolicy / AuditLog** 35分

1 Base クラスとエンジン設定

Claude Code に次のように依頼します。

PROMPT

```
app/models/base.py に DeclarativeBase を継承した Base クラスを実装してください。
app/database.py を非同期エンジン (aiosqlite) 対応に更新し、AsyncSession を返す get_session を提供してください。
.github/instructions/sqlalchemy.instructions.md の規約を必ず読んでから書いてください。
```

2 エンティティ実装の指示

PROMPT

```
app/models/ 配下に下記6エンティティを SQLAlchemy 2.0の Mapped 構文で実装してください。
docs/data_model.md の ER図を正としてカラムと型を決めます。

- User (id, name, email[unique], role, created_at)
- Category (id, name[unique], description)
- Ticket (id, title, description, status, priority, reporter_id[FK], assignee_id[FK,nullable], category_id[FK], created_at, updated_at, resolved_at[nullable])
- Comment (id, ticket_id[FK], author_id[FK], body, created_at)
- SLAPolicy (id, priority[unique], response_minutes, resolution_minutes)
- AuditLog (id, actor_id[FK], action, target_type, target_id, before_json, after_json, created_at)

すべてのカラムに型ヒント (Mapped[T])、PK/FK 制約、デフォルト値を明示してください。
created_at は datetime(timezone=True) で UTC 保存。
Ticket.status と Ticket.priority は str 型で保存しますが、enum を定義して値を制約する関数を schemas/ticket.py 側で書きます。
```

3 シードデータの整備

data/seed/initial_tickets.json は配布物に既に入っています。これを起動時に DB に流し込むスクリプトを書きます。

PROMPT

```
scripts/seed.py を作り、data/seed/initial_tickets.json を読んで users, categories, tickets, comments, sla_policies を順に DB に投入してください。
SLA policy はヒアリングメモの値 (high 30分/4h、medium 4h/1日、low 1日/3日) を初期値とします。
冪等性のため、起動毎に DELETE → INSERT する形でOKです。
```

4 動作確認

```
python scripts/seed.py
sqlite3 helpdesk.db "SELECT COUNT(*) FROM tickets;"
sqlite3 helpdesk.db "SELECT id, name, role FROM users;"
```
TERMINAL

チケット数とユーザー一覧が想定通り表示されればOK。

よくある詰まり

- `ImportError: cannot import name 'Mapped'` → SQLAlchemy 2.0系が入っていない。`pip show sqlalchemy` で2.0系か確認
- 非同期エンジンでブロッキング呼び出し → `create_engine` ではなく `create_async_engine` を使う
- FK 制約違反 → SQLite は外部キー制約が既定で無効。`PRAGMA foreign_keys=ON` を有効化

SESSION 04 ／ 14:00〜14:15

ユニットテストの追加

repository 層のユニットテストでカバレッジの土台を作ります。15分。

HANDS-ON

repository ユニットテスト

15分

1 非同期 fixture の整備

tests/conftest.py を更新し、in-memory SQLite の非同期セッションを返す fixture を追加します。

PROMPT

```
tests/conftest.py を更新し、下記の fixture を追加してください。

- async_engine: aiosqlite で in-memory な AsyncEngine
- async_session: 上記から作る AsyncSession（テスト毎にロールバック）
- seed_session: async_session に最小限のシードデータを入れた状態を返す

pytest-asyncio の asyncio_mode = "auto" 設定済みなので、async def テストはそのまま動きます。
```

2 ticket_repository のユニットテストを生成

Claude Code に厳密モード (strict-reviewer) で依頼します。

PROMPT

```
/agents strict-reviewer

tests/unit/test_ticket_repository.py を作り、ticket_repository の各メソッドに対して
正常系・境界値・異常系のテストを最低3ケースずつ書いてください。
.github/prompts/generate-tests.prompt.md の規約に従ってください。
```

3 カバレッジ計測

```
pytest tests/ --cov=app --cov-report=term-missing
```
TERMINAL

app/repositories/ticket_repository.py のカバレッジが80%以上、app/models/ticket.py が60%以上になるのが目安です。

休憩(14:15〜14:25)

SESSION 05 ／ 14:25〜15:00

API層の実装

35分で router → service → repository のレイヤを全エンドポイント分積み上げます。

HANDS-ON レイヤ別にAPIを実装

35分

1 エラーレスポンス共通化

最初にエラーレスポンスの共通形を実装します。これで以降の実装が楽になります。

PROMPT

```
app/schemas/errors.py に ErrorResponse スキーマを実装してください。
{"error": {"code": str, "message": str}} の形式です。
続いて app/exceptions.py にドメイン例外 (TicketNotFound, InvalidStatusTra
nsition, SLAPolicyNotFound) を定義し、
app/main.py に FastAPI の exception_handler を登録して、これらの例外を統一形
式のレスポンスに変換してください。
```

2 チケット CRUD の実装

PROMPT

```
app/repositories/ticket_repository.py に CRUD を実装してください
(list_by_filter, get_by_id, create, update, delete, search_by_text)。
app/services/ticket_service.py で状態遷移ルールを検証する update_status を
実装し、
無効な遷移は InvalidStatusTransition を raise してください。
app/routers/tickets.py で router を整備し、FastAPI の Depends で AsyncSe
ssion を注入します。
レスポンス・リクエストスキーマは app/schemas/ticket.py に定義済みのものを使い
ます。
```

3 SLA計算サービス

SLA は本研修の中で最も複雑なロジックです。営業時間外と土日祝の除外を考慮します。

PROMPT

```
app/services/sla_service.py に下記を実装してください。

1. calculate_response_deadline(ticket): チケットの優先度に応じた初回応答期限を計算
2. calculate_resolution_deadline(ticket): 解決期限
3. is_response_breached(ticket): 初回応答 SLA が違反しているか判定
4. is_resolution_breached(ticket): 解決 SLA が違反しているか判定
5.営业時間外 (19:00-9:00) と土日祝は計算から除外する
6. list_breached_tickets(session): 違反中の全チケットを返す

祝日カレンダーは Phase 2のスコープなので、当面は土日のみ除外で OK です。
ロジックを変更しやすいよう、is_business_hour(datetime) を分離してください。
```

---

4 コメント API の実装

PROMPT

```
app/routers/comments.py と app/services/comment_service.py と app/repositories/comment_repository.py を実装してください。
コメント本文は最大2000文字、空文字不可とバリデーションします。
HTMLタグはエスケープせず生で保存しますが、レスポンス時にエスケープを適用します。
add_comment 時に AuditLog レコードも書き込んでください。
```

---

5 CIを緑のまま保つ

ここまでで一度コミット・push。CIの4ジョブが全て緑になることを確認してください。
落ちたら原因を Chat に貼って直してから先に進みます。

SESSION 06 ／ 15:00〜15:30

結合テストとカバレッジ80%

30分で TestClient ベースの結合テストを書き、カバレッジ80%を達成します。

HANDS-ON API結合テスト

30分

1 TestClient フィクスチャの整備

tests/conftest.py に async TestClient を返す fixture を追加します。get_session の依存を上書きして、テスト用 in-memory DB に切り替えます。

2 結合テストの生成

```
PROMPT">/AGENTS STRICT-REVIEWER
```

3 SLA計算のテスト

SLA は時刻依存のロジックなので、fixture で時刻を固定します。

PROMPT

```
tests/unit/test_sla_service.py を作り、SLA計算ロジックの境界値テストを書いてください。
- 平日9:00開始のhighチケットが30分後に未応答だったら違反
- 営業時間外を跨ぐと営業時間分のみカウントされる
- 土日を跨ぐと土日は除外される
- 各優先度の境界値ぴったりで違反/未違反が切り替わる

freezegun でテスト内の時刻を固定してください（依存に追加する必要があれば requirements-dev.txt も更新）。
```

4 **カバレッジ80%達成確認**

```
pytest tests/ --cov=app --cov-report=term-missing --cov-fail-under=80
```
TERMINAL

カバレッジが80%未満なら CI が失敗します。未カバーの行を見て、優先度の高い箇所からテストを追加してください。

SESSION 07 ／ 15:30〜15:50

動作確認とコードレビュー

Swagger UI で実機確認し、Strict Reviewer で全体を点検する20分です。

HANDS-ON **動作確認 + 設計レビュー** 15分

1 ローカルでアプリ起動

```
uvicorn app.main:app --reload --port 8000
```
TERMINAL

ブラウザで `http://127.0.0.1:8000/docs` を開き、自分が実装した全エンドポイントを Swagger UI から叩いて挙動を確認します。

2 Day1で起こした公開デモと比較

https://jbs-0514.vercel.app/docs を別タブで開き、自分の実装と挙動が一致しているか比較してください。違いがあれば、公開デモは「仕込まれた改善余地を含む雑な実装」だったことを思い出してください。自分の実装は規約通りに動くはずです。

3 Strict Reviewer で全体レビュー

```
/agents strict-reviewer

app/ 配下のコードを全部読み、下記の観点で点検してください。
1. routers → services → repositories の依存方向が守られているか
2. エラーレスポンスが全エンドポイント統一されているか
3. 入力バリデーション（長さ、空文字、型）に漏れがないか
4. SQLAlchemy 2.0の Mapped 構文を使えているか
5. SQL injection や認可漏れの余地がないか
6. テストが正常系・境界値・異常系の3観点を満たしているか
```
PROMPT

4 致命的な指摘のみ反映

Strict Reviewer は妥協なく指摘してきますが、Day2内で全て直す必要はありません。「致命的」とラベル付けされたものだけ即修正し、それ以外は Issue として GitHub に起票して Day3で対応します。

5 commit と push

```
git add app/ tests/
git commit -m "feat: Day2完了 アプリ実装とテスト"
git push
```
TERMINAL

Actions タブで CI が緑になっていることを確認したら Day2完了です。

CLOSING ／ 15:50〜16:00

まとめとDay3への引き継ぎ

Day2の成果物と、Day3までの宿題を整理します。

Day2の成果物チェック

- [x] app/ 配下に routers / services / repositories / models / schemas / exceptions が揃っている
- [x] pytest 全件 PASS、カバレッジ80%以上
- [x] ruff check と mypy が違反ゼロ
- [x] Swagger UI で全エンドポイントが動作
- [x] CI(4ジョブ)が緑、git push 済み

Day3までの宿題

- Strict Reviewer が出した「中」「軽微」の指摘から1〜2件を Issue 化、自分で直してみる
- Dockerfile の概念に不安があれば、配布物の `Dockerfile` を読んでマルチステージビルドの構造を理解しておく
- 余裕があれば `docker-compose up` をローカルで試してみる(Day3で扱う内容ですが、事前に動かしておくと理解が早い)

本日覚えておきたい3点

- [x] レイヤ単方向依存(routers → services → repositories)を守ると、テストが書きやすくなる
- [x] テストは実装と同時に書く。後回しにすると Day3の CI で破綻する
- [x] Strict Reviewer のような厳しい視点を切り替えて使えると、人間レビューの密度が上がる

> **Day3のフォーカス予告**
>
> Day3(5/28)は本日の実装を Dockerfile マルチステージビルドでパッケージ化し、GitHub Actions の CI 全ジョブを緑のまま docker build を追加します。`@claude` メンションで AI 自動 PR 修正も体験。事前に `.github/prompts/dockerfile-review.prompt.md` に目を通しておくとスムーズです。

制作 Givery／JBS 中級コース 案1 Day2配布用／2026-05-21