

注文管理システム 実装ガイド

この実装ガイドでは、Todoアプリチュートリアルで学んだ知識を応用して、注文管理システムを作成する手順を解説します。

機能ごとにバックエンド→フロントエンドの順で実装し、動作確認しながら進めます。

前提条件

- Next.js + FastAPI + PostgreSQL Todoアプリチュートリアル を完了している
- Node.js、Python、Docker がインストールされている

学習パス: [Next.js フルスタック学習パス](#)

推奨所要時間

ステップ別（バックエンド・フロントエンド内訳付き）

| ステップ | 内容 | バックエンド | フロントエンド | 合計 |
|--------|---------|---------|----------|------------|
| Step 1 | 環境構築 | 4~6時間 | 4~6時間 | 8~12時間 |
| Step 2 | 顧客管理 | 8~12時間 | 12~18時間 | 20~30時間 |
| Step 3 | 商品管理 | 6~10時間 | 10~16時間 | 16~26時間 |
| Step 4 | 注文管理 | 16~24時間 | 24~36時間 | 40~60時間 |
| Step 5 | ダッシュボード | 6~10時間 | 10~14時間 | 16~24時間 |
| Step 6 | 仕上げ | 4~8時間 | 6~10時間 | 10~18時間 |
| 合計 | | 44~70時間 | 66~100時間 | 約110~170時間 |

実装コード量の目安

| 領域 | 行数 | 主な内容 |
|---------|---------|---------------------|
| バックエンド | 約630行 | モデル4種、API 4エンドポイント群 |
| フロントエンド | 約2,150行 | 画面11種、コンポーネント3種 |

時間かかるポイント

- Step 4（注文管理）が最も難しい: リレーション、JOIN、動的フォーム、合計計算など新しい概念が多い
- フロントエンドはバックエンドの約1.5倍: UI実装、状態管理、エラー処理に時間がかかる
- デバッグ時間: 実装時間と同程度のデバッグ時間を見込むこと

※ 初めて取り組む方は、調べながらの作業やエラー対応で想定以上に時間がかかることがあります。焦らず余裕を持って進めてください。1日3~4時間の学習で、約1~2ヶ月を目安にしてください。

Step 1: 環境構築

1-1. プロジェクト構成

```
order-management/
├── docker-compose.yml
└── backend/
    ├── app/
    │   ├── __init__.py
    │   ├── main.py
    │   ├── database.py
    │   └── models/
    │       └── __init__.py
    └── routers/
        └── __init__.py
    └── requirements.txt
    └── venv/
└── frontend/
    └── (Next.jsプロジェクト)
```

1-2. やること

1. プロジェクトフォルダ `order-management` を作成

2. `docker-compose.yml` を作成 (PostgreSQL設定)
3. `backend` フォルダを作成し、Python環境をセットアップ
4. `frontend` フォルダに Next.js プロジェクトを作成

1-3. データベース設定

- データベース名: `order_db`
- ユーザー名/パスワード: `postgres / postgres`
- ポート: `5432`

1-4. バックエンド必要パッケージ

- `fastapi`
- `uvicorn[standard]`
- `sqlmodel`
- `psycopg2-binary`

1-5. フロントエンド設定

Next.js 作成時の選択:

- TypeScript: Yes
- Tailwind CSS: Yes
- App Router: Yes
- src/ directory: Yes

1-6. 共通ファイルの作成

バックエンド

- `database.py` : データベース接続設定
- `main.py` : FastAPIアプリケーション (CORS設定含む)

フロントエンド

- `src/types/index.ts` : 型定義 (Customer, Product, Order 等)
- `src/lib/api.ts` : API通信の共通関数
- `src/components/Header.tsx` : ナビゲーションヘッダー

- `src/app/layout.tsx` : 共通レイアウト

1-7. 確認

- `docker compose up -d` でデータベース起動
- `uvicorn app.main:app --reload` でバックエンド起動 → <http://localhost:8000/docs>
- `npm run dev` でフロントエンド起動 → <http://localhost:3000>

ヒント

Todoアプリチュートリアルの Step 1~3, 6 を参考にしてください。

Step 2: 顧客管理機能

2-1. 概要

顧客情報（名前、メール、電話番号）を管理する機能を実装します。Todoアプリで学んだCRUD操作の復習として最適です。

2-2. バックエンド実装

モデル作成

`app/models/customer.py` を作成

| フィールド | 型 | 必須 | 説明 |
|-------------------------|-----------------------|-----------------------|--------------|
| <code>id</code> | <code>int</code> | 自動 | 主キー |
| <code>name</code> | <code>str</code> | <input type="radio"/> | 顧客名（最大100文字） |
| <code>email</code> | <code>str</code> | - | メールアドレス |
| <code>phone</code> | <code>str</code> | - | 電話番号 |
| <code>created_at</code> | <code>datetime</code> | 自動 | 作成日時 |
| <code>updated_at</code> | <code>datetime</code> | 自動 | 更新日時 |

API実装

app/routers/customers.py を作成

| メソッド | パス | 機能 |
|--------|---------------------|---------------------------|
| GET | /api/customers | 一覧取得 (search パラメータで名前検索) |
| GET | /api/customers/{id} | 詳細取得 |
| POST | /api/customers | 新規登録 |
| PUT | /api/customers/{id} | 更新 |
| DELETE | /api/customers/{id} | 削除 |

実装ポイント

- 検索は名前の部分一致 (`contains`)
- 削除は後で注文機能を作った後に「注文があれば削除不可」を追加

確認

Swagger UI (<http://localhost:8000/docs>) で各APIをテスト

2-3. フロントエンド実装

画面一覧

| パス | ファイル | 機能 |
|----------------------|----------------------------------|----------|
| /customers | app/customers/page.tsx | 一覧・検索・削除 |
| /customers/new | app/customers/new/page.tsx | 新規登録フォーム |
| /customers/[id]/edit | app/customers/[id]/edit/page.tsx | 編集フォーム |

共通コンポーネント

components/CustomerForm.tsx : 登録・編集で共通のフォーム

実装ポイント

- 一覧: `useState` + `useEffect` でAPI呼び出し、テーブル表示
- 登録/編集: フォーム状態管理、送信後に一覧ヘリダイレクト
- バリデーション: 名前は必須

確認

- 顧客を登録できる
- 一覧に表示される
- 編集・削除ができる
- 名前で検索できる

ヒント

Todoアプリで作った一覧・追加機能を、複数項目のフォームに拡張します。

Step 3: 商品管理機能

3-1. 概要

商品情報（名前、価格、カテゴリ、販売状態）を管理する機能を実装します。顧客管理とほぼ同じパターンで実装できます。

3-2. バックエンド実装

モデル作成

`app/models/product.py` を作成

| フィールド | 型 | 必須 | 説明 |
|--------------|----------|----|-------------------------|
| id | int | 自動 | 主キー |
| name | str | ○ | 商品名 (最大100文字) |
| price | int | ○ | 価格 (1~999999) |
| category | str | ○ | カテゴリ (food/drink/other) |
| is_available | bool | - | 販売可能 (デフォルト: true) |
| created_at | datetime | 自動 | 作成日時 |
| updated_at | datetime | 自動 | 更新日時 |

API実装

app/routers/products.py を作成

| メソッド | パス | 機能 |
|--------|--------------------|------------------------------------|
| GET | /api/products | 一覧取得 (category , available でフィルタ) |
| GET | /api/products/{id} | 詳細取得 |
| POST | /api/products | 新規登録 |
| PUT | /api/products/{id} | 更新 |
| DELETE | /api/products/{id} | 削除 |

実装ポイント

- 価格のバリデーション: `Field(ge=1, le=999999)`
- カテゴリでフィルタ可能に

3-3. フロントエンド実装

画面一覧

| パス | 機能 |
|---------------------|--------------------|
| /products | 一覧・カテゴリフィルタ・販売状態切替 |
| /products/new | 新規登録フォーム |
| /products/[id]/edit | 編集フォーム |

実装ポイント

- カテゴリフィルタ: ボタンまたはタブで切り替え
- 販売状態: トグルボタンで切り替え (APIをPUT)
- 価格表示: `toLocaleString()` で3桁カンマ

確認

- 商品を登録できる
- カテゴリでフィルタできる
- 販売中/停止を切り替えられる

Step 4: 注文管理機能

4-1. 概要

顧客と商品を組み合わせて注文を作成・管理する機能です。複数テーブルのリレーションとステータス管理を学びます。

4-2. バックエンド実装

モデル作成

Order (注文ヘッダー) : `app/models/order.py`

| フィールド | 型 | 必須 | 説明 |
|--------------|----------|----|------------------------|
| id | int | 自動 | 主キー |
| customer_id | int | ○ | 顧客ID (外部キー) |
| status | str | - | ステータス (デフォルト: pending) |
| total_amount | int | ○ | 合計金額 |
| created_at | datetime | 自動 | 注文日時 |
| updated_at | datetime | 自動 | 更新日時 |

OrderItem (注文明細) : app/models/order_item.py

| フィールド | 型 | 必須 | 説明 |
|------------|-----|----|-------------|
| id | int | 自動 | 主キー |
| order_id | int | ○ | 注文ID (外部キー) |
| product_id | int | ○ | 商品ID (外部キー) |
| quantity | int | ○ | 数量 (1~99) |
| unit_price | int | ○ | 注文時の単価 |
| subtotal | int | ○ | 小計 |

API実装

app/routers/orders.py を作成

| メソッド | パス | 機能 |
|--------|-------------------------|-------------------|
| GET | /api/orders | 一覧取得 (ステータスでフィルタ) |
| GET | /api/orders/{id} | 詳細取得 (明細含む) |
| POST | /api/orders | 新規作成 |
| PUT | /api/orders/{id}/status | ステータス更新 |
| DELETE | /api/orders/{id} | キャンセル |

注文作成リクエスト形式

```
{  
  "customer_id": 1,  
  "items": [  
    { "product_id": 1, "quantity": 2 },  
    { "product_id": 3, "quantity": 1 }  
  ]  
}
```

注文作成の処理フロー

1. 顧客の存在確認
2. 各商品の存在・販売状態確認
3. 単価取得 → 小計計算 → 合計計算
4. 注文ヘッダー作成
5. 注文明細作成

ステータス遷移ルール

```
pending (受付)  
  ↓ cooking または cancelled  
cooking (調理中)  
  ↓ completed または cancelled  
completed (完了) ← 終了状態  
cancelled (キャンセル) ← 終了状態
```

JOINの使用

一覧・詳細で顧客名・商品名を取得するため、JOINが必要:

```
select(Order, Customer.name).join(Customer, Order.customer_id == Customer.id)
```

4-3. フロントエンド実装

画面一覧

| パス | 機能 |
|--------------|----------------------|
| /orders | 一覧・ステータスフィルタ |
| /orders/new | 注文作成（顧客選択、商品追加、数量変更） |
| /orders/[id] | 詳細表示・ステータス更新 |

注文作成画面の実装ポイント

1. 顧客選択: ドロップダウン (`<select>`)
2. 商品追加: 商品一覧から選択 → 注文リストに追加
3. 数量変更: +/- ボタンまたは数値入力
4. 合計計算: 明細の小計を合計 (リアルタイム更新)
5. 注文確定: 送信後に一覧ヘリダイレクト

注文詳細画面の実装ポイント

1. 明細表示: 商品名、数量、単価、小計をテーブル表示
2. ステータス表示: 色分け (受付:黄、調理中:青、完了:緑、キャンセル:灰)
3. 操作ボタン: 次のステータスへの遷移ボタン

確認

- 顧客を選択して注文を作成できる
- 商品を追加・削除、数量を変更できる
- 合計金額が正しく計算される
- ステータスを更新できる

ヒント

- [database.md](#) の「注文作成の処理フロー」を参照
- [README.md](#) の「注文ステータス」を参照

Step 5: ダッシュボード機能

5-1. 概要

本日の売上サマリーと最近の注文を表示するトップページです。集計クエリ (COUNT, SUM) を学びます。

5-2. バックエンド実装

API実装

`app/routers/dashboard.py` を作成

| メソッド | パス | 機能 |
|------|---|------------------|
| GET | <code>/api/dashboard/summary</code> | 本日のサマリー |
| GET | <code>/api/dashboard/recent-orders</code> | 最近の注文 (デフォルト10件) |

サマリーのレスポンス

| 項目 | 説明 |
|-----------------------------|----------------------------|
| <code>today_orders</code> | 本日の注文件数 (キャンセル除く) |
| <code>today_sales</code> | 本日の売上合計 |
| <code>pending_orders</code> | 未完了注文数 (pending + cooking) |
| <code>status_summary</code> | ステータス別件数 |

実装ポイント

- 集計関数: `func.count()`, `func.sum()`
- 日付フィルタ: `func.date(Order.created_at) == date.today()`
- NULL対策: `func.coalesce(func.sum(...), 0)`

5-3. フロントエンド実装

画面

app/page.tsx (トップページ)

表示内容

1. サマリーカード: 本日の注文数、売上、未完了注文
2. ステータス別件数: バッジまたはチップで表示
3. 最近の注文一覧: テーブル (詳細へのリンク付き)
4. 新規注文ボタン: /orders/new へのリンク

確認

- 本日の注文・売上が正しく表示される
- 注文を追加すると数値が更新される

Step 6: 仕上げ

6-1. 削除制約の実装

顧客・商品の削除時に、関連する注文があれば削除不可にする。

顧客削除 (customers.py):

- 注文テーブルに該当customer_idがあればエラー

商品削除 (products.py):

- 注文明細テーブルに該当product_idがあればエラー

6-2. エラーハンドリング

- APIエラー時にユーザーへメッセージ表示
- 404 (データなし) 時の表示
- バリデーションエラーの表示

6-3. 動作確認チェックリスト

- 顧客を登録・編集・検索・削除できる

- 商品を登録・編集・カテゴリフィルタ・販売切替できる
- 注文を作成できる（顧客選択、商品追加、数量変更）
- 注文のステータスを更新できる
- ダッシュボードに売上サマリーが表示される
- 注文がある顧客・商品は削除できない

6-4. 追加課題（余力があれば）

- ローディング表示
 - レスポンシブデザイン
 - 期間指定の売上検索
 - 注文履歴の日付フィルタ
-

完成後のスキル

- 複数テーブルを持つデータベース設計
 - 外部キー制約とリレーションの理解
 - JOINを使った複合クエリの作成
 - ステータス管理を含むビジネスロジックの実装
 - 動的なフォーム（明細行の追加・削除）の実装
 - 集計機能とダッシュボード画面の作成
-

困ったときは

1. Todoアプリチュートリアルを見直す
2. [database.md](#) でテーブル設計・SQLを確認
3. [README.md](#) でAPI仕様を確認
4. 学習ガイドで各技術の詳細を学ぶ:
 - [FastAPI学習ガイド](#)
 - [SQL入門学習ガイド](#)
 - [React学習ガイド](#)

頑張ってください！