# データベース最終課題
# プレゼン資料

2442005 五十嵐玲有
2442012 岩本光多郎
2442082 三浦篤史
2442083 村瀬優直
2442087 山口漣麗

https://github.com/graceripple/Burg-burg

# アプリの実装コード

```python
"""
講義内容の実装:
- 第4回: トランザクション処理・ACID特性
- 第3回: 関係演算（選択・射影・結合）
- 第11回: 正規化されたデータベース設計
"""

from flask import Flask, jsonify, request, render_template_string
from database import get_db_cursor, execute_transaction, init_connection_pool
import os
from dotenv import load_dotenv

load_dotenv()

app = Flask(__name__)

# データベース接続プール初期化
init_connection_pool()


# ======================================
# API: 商品一覧取得
# 第3回: SELECT演算（選択・射影）
# ======================================
            # カラム名を取得
            columns = [desc[0] for desc in cursor.description]
            # 結果を辞書形式に変換
            products = [dict(zip(columns, row)) for row in cursor.fetchall()]

            return jsonify({'success': True, 'data': products})

    except Exception as e:
        print(f'Error fetching products: {e}')
        return jsonify({'success': False, 'error': 'Failed to fetch products'}), 500


# ======================================
# API: 商品詳細取得
# ======================================
@app.route('/api/products/<int:product_id>', methods=['GET'])
def get_product(product_id):
    """特定の商品の詳細を取得"""
    try:
        with get_db_cursor() as cursor:
            cursor.execute("""
                SELECT * FROM products WHERE product_id = %s
            """, (product_id,))

    except Exception as e:
        print(f'Error fetching product: {e}')
        return jsonify({'success': False, 'error': 'Failed to fetch product'}), 500


# ======================================
# API: 注文作成（トランザクション処理）
# 第4回: ACID特性の実装例
# ======================================
@app.route('/api/orders', methods=['POST'])
def create_order():
    """
    注文を作成する（トランザクション処理）

    講義第4回の送金例と同じ構造:
    1. 在庫チェック（ロック取得）
    2. 注文レコード作成
    3. 注文明細作成
    4. 在庫更新
    5. 在庫履歴記録
    6. 合計金額更新

    エラー時は自動的にロールバック（Atomicity: 原子性）
    """
    try:
        if not customer_name or not items:
            return jsonify({
                'success': False,
                'error': 'Invalid request: customer_name and items are required'
            }), 400

        # トランザクション実行
        def order_transaction(cursor):
            # 1. 在庫チェック（ロックを取得）
            for item in items:
                cursor.execute("""
                    SELECT product_id, product_name, stock_quantity, unit_price
                    FROM products
                    WHERE product_id = %s
                    FOR UPDATE  -- 行レベルロック取得（Isolation: 隔離性）
                """, (item['product_id'],))

                product = cursor.fetchone()
                if not product:
                    raise Exception(f"商品ID {item['product_id']} が見つかりません")

                if product[2] < item['quantity']:
                    raise Exception(
                        f"商品「{product[1]}」の在庫が不足しています。"
                        f"（在庫: {product[2]}個、注文: {item['quantity']}個）"
            # 2. 注文レコード作成
            cursor.execute("""
                INSERT INTO orders (customer_name, total_amount, status)
                VALUES (%s, 0, 'pending')
                RETURNING order_id
            """, (customer_name,))

            order_id = cursor.fetchone()[0]
            total_amount = 0

            # 3. 注文明細作成と在庫更新
            for idx, item in enumerate(items, 1):
                # 商品情報取得
                cursor.execute("""
                    SELECT unit_price, stock_quantity FROM products
                    WHERE product_id = %s
                """, (item['product_id'],))

                unit_price, stock_before = cursor.fetchone()

                # 注文明細追加
                cursor.execute("""
                    INSERT INTO order_details (order_id, detail_number, product_id, quantity, unit_price)
                    VALUES (%s, %s, %s, %s, %s)
                """, (order_id, idx, item['product_id'], item['quantity'], unit_price))

                # 在庫更新
                stock_after = stock_before - item['quantity']

                # 在庫履歴記録
                cursor.execute("""
                    INSERT INTO stock_history (product_id, change_type, quantity_change, stock_before, stock_after, order_id)
                    VALUES (%s, 'sale', %s, %s, %s, %s)
                """, (item['product_id'], -item['quantity'], stock_before, stock_after, order_id))

                total_amount += unit_price * item['quantity']

            # 4. 注文合計金額更新
            cursor.execute("""
                UPDATE orders
                SET total_amount = %s,
                    status = 'completed'
                WHERE order_id = %s
            """, (total_amount, order_id))

            return {'orderId': order_id, 'totalAmount': float(total_amount)}

        # トランザクション実行
        result = execute_transaction(order_transaction)

        return jsonify({
            'success': True,
            'message': '注文が完了しました',
            'data': result
        })

    except Exception as e:
        print(f'Order creation error: {e}')
        return jsonify({
            'success': False,
            'error': str(e)
        }), 400
```

# 実装コード②

```python
# ====================================
# API: 注文一覧取得
# ====================================
@app.route('/api/orders', methods=['GET'])
def get_orders():
    """注文履歴一覧を取得"""
    try:
        with get_db_cursor() as cursor:
            cursor.execute("""
                SELECT
                    order_id,
                    customer_name,
                    order_date,
                    total_amount,
                    status
                FROM orders
                ORDER BY order_date DESC
                LIMIT 50
            """)

            columns = [desc[0] for desc in cursor.description]
            orders = [dict(zip(columns, row)) for row in cursor.fetchall()]

            return jsonify({'success': True, 'data': orders})

    except Exception as e:
        print(f'Error fetching orders: {e}')
        return jsonify({'success': False, 'error': 'Failed to fetch orders'}), 500
```

```python
# 担当: JOIN演算（結合）
# ====================================
@app.route('/api/orders/<int:order_id>', methods=['GET'])
def get_order(order_id):
    """注文詳細を取得（商品情報を結合）"""
    try:
        with get_db_cursor() as cursor:
            # 注文ヘッダ取得
            cursor.execute("""
                SELECT * FROM orders WHERE order_id = %s
            """, (order_id,))

            row = cursor.fetchone()
            if not row:
                return jsonify({'success': False, 'error': 'Order not found'}), 404

            columns = [desc[0] for desc in cursor.description]
            order = dict(zip(columns, row))

            # 注文明細取得（商品情報を結合）
            cursor.execute("""
                SELECT
                    od.*,
                    p.product_name
                FROM order_details od
                INNER JOIN products p ON od.product_id = p.product_id
                WHERE od.order_id = %s
                ORDER BY od.detail_number
            """, (order_id,))

            columns = [desc[0] for desc in cursor.description]
            details = [dict(zip(columns, row)) for row in cursor.fetchall()]

            order['details'] = details

            return jsonify({'success': True, 'data': order})

    except Exception as e:
        print(f'Error fetching order: {e}')
        return jsonify({'success': False, 'error': 'Failed to fetch order'}), 500
```

```python
# ====================================
# フロントエンド: メインページ
# ====================================
@app.route('/')
def index():
    """メインページを表示"""
    html = open('/home/user/webapp-python/templates/index.html', 'r', encoding='utf-8').read()
    return html


if __name__ == "__main__":
    # host="0.0.0.0" が重要です！これがないとブラウザから見えません。
    app.run(host="0.0.0.0", port=3000, debug=True)

```

# アプリのデモ動画


在庫商品一覧
ノートパソコン
在庫: 8
¥89,800
商品ID: 1
カートに追加
ワイヤレスマウス
在庫: 47
¥2,980
商品ID: 2
カートに追加
USB-Cケーブル
在庫: 100
¥1,280
商品ID: 3
カートに追加
モニターアーム
在庫: 15
¥8,900
商品ID: 4
カートに追加

# ペルソナ定義

- 名前：中村裕子（３４歳）
- 職業：個人経営の電化製品の店
- 人物像：20～50代、店舗運営の実務担当（レジ、発注、在庫確認、簡単な経理も兼任しがち）、専任の情シス/データ担当はいない、“正しい入力”よりも“止まらず回る”ことが最重要（忙しい時間帯に使う）
- 達成したいこと（Goals / Jobs）：いま売れる在庫があるかを即確認したい、注文を受けたら、迷わず最短で登録して確定したい、過去の取引（いつ・誰に・何を・いくらで）をすぐ見返したい
- 困りごと（Pains）：在庫が実態とズレる（販売登録漏れ、二重入力、返品の反映漏れ）、商品名が曖昧で検索しづらい（似た商品が多い）、入力項目が多いと運用が破綻する（忙しい時に入力できない）、合計金額の計算ミスが怖い（割引/税/端数など）
- 利用状況（Context）：ピーク時：1件あたり30～60秒で処理したい、PC/タブレット中心、画面は広くないこともある、“紙や口頭の注文”→“あとでまとめ入力”も起きる

# プロジェクトの目的と概要

開発目的: 効率的な販売管理を実現するためのWebアプリケーション構築

主要機能（動画の実装内容）:

・在庫商品の一覧表示と管理

・新規注文の作成（顧客名入力、商品選択、数量指定、合計金額の自動計算）

・注文履歴の保持と詳細情報の確認機能

# ビジネス要件定義

ターゲット（ペルソナ）：小規模な店舗の在庫・販売管理を簡略化したい担当者

ストーリーボード:

１．担当者が現在の在庫一覧を確認する。

２．新規注文を受け、顧客名と商品を入力する。

３．システムが自動計算した合計金額を確認し、注文を確定させる。

４．履歴画面で過去の取引内容を振り返る 。

# システムアーキテクチャ

構成: 標準的な Web 3 Layer 構成を採用

構成要素:

・Web/Client層: ブラウザベースのユーザーインターフェース（動画内の操作画面）

・Application層: Python (Flask等) を利用したビジネスロジック処理

・Database層: RDB を用いた永続的なデータ管理

# データベース設計

データモデル: RDB を利用し、ER図に基づいたテーブル設計を実施

主なエンティティ:

- Users (注文者/顧客)
- Products (在庫商品：ノートパソコン、ワイヤレスマウス等)
- Orders (注文履歴：ID、注文日時、合計金額)

CRUD操作の実装: 在庫の読み取り (Read) と新規注文の登録 (Create) を網羅

# 実装機能の解説（デモ動画に基づく）

## 在庫商品一覧画面:

- 商品名、価格、在庫数、および個別の画像を表示。

## 新規注文プロセス:

- ドロップダウンリストから商品を選択（例：HDMIケーブル 2m ￥1,580）。
- 数量の増減に応じた小計・合計のリアルタイム更新。

## 注文履歴と詳細:

- 「注文を確定」ボタン押下後、履歴一覧へ即時反映。
- ステータス管理（「完了」等）とモーダルによる詳細表示機能。

# インフラ構成と非機能要件

環境構築: Webサーバー、アプリケーションサーバー、DBの連携設定

環境変数管理: .env ファイルを用いたセキュアなDB接続情報の保持

今後の課題: RPO/RTO（目標復旧時間）の定義やバックアップ体制、パフォーマンスの最適化

# プロジェクト管理と進捗

GitHub Project の活用: Kanban Board を用いたタスク・チケット管理

開発手法: Scrum マスターを中心とした Sprint 管理による段階的な実装

# まとめ

## 本プロジェクトの成果

- 全工程の完遂: ビジネス要件定義からインフラ構築、アプリケーション実装に至る開発工程を統合的に遂行した 。
- Web 3層構造の実装: FlaskとRDBを用い、保守性の高い3層構造アーキテクチャを実現した 。
- 実用的な販売管理機能: 動画の通り、在庫確認から注文登録、履歴管理までの主要なビジネスロジックを具現化した。

# まとめ

## 技術的習得事項

- アジャイル管理の実践: GitHub ProjectとKanban Boardを活用し、タスクの可視化と工程管理の重要性を習得した 。

- 実務的設計視点: .envによるセキュリティ管理や、RPO/RTOを意識した非機能要件の設計手法を理解した 。

# まとめ

## 今後の課題

- 機能の高度化: 在庫の自動減算処理や売上統計機能の実装による、利便性の向上を図る。

- 運用の自動化: CI/CDパイプラインの導入により、開発からデプロイまでの効率化を検討する。