ChatGPT

FlaskとDockerで学ぶStripeGymカリキュラム (Checkout + Webhook + 台帳)

Stripe未経験者を対象に、**Stripe　Checkout**による決済、**Webhook**受信、そして自前の**台帳**（トランザクション記録）の実装を、Flaskアプリケーションと Docker環境で段階的に解説します。小さく作ってテストし、記録とログを検証しながら進める学習サイクルを意識しています。各フェーズ（Day0〜Day6）ごとにゴールとタスクを明確にし、必要なコード例やStripe側の設定手順、確認方法、トラブルシュート、次段階へ進む条件を整理しています。最終的にプロジェクトのREADMEテンプレートも提示します。

初動計画 (Day0〜Day6)

- **Day0**: 開発環境の構築（Flaskプロジェクト作成、Dockerで実行できる状態に）
- **Day1**: Stripe Checkoutの導入（Stripe APIキー設定、Checkout Session作成と決済開始）
- **Day2**: 決済フローの確認とUI整備（成功/キャンセルページの実装、テスト決済の検証とStripeダッシュボード確認）
- **Day3**: Webhook受信の実装（FlaskでWebhookエンドポイントを作成し、Stripeイベントを受信）
- **Day4**: 台帳（Ledger）の実装（Webhookで受け取った支払い情報をデータベースに記録）
- **Day5**: 総合テストとトラブルシューティング（複数回の決済で台帳とStripe記録の突合、ログ確認、問題解決）
- **Day6**: ドキュメンテーションと次ステップ（README整備、本番運用に向けた設定確認など）

Day0: 環境準備とプロジェクトセットアップ

ゴール: Flaskアプリの雛形を作成し、Dockerコンテナ上で「Hello, World」を表示できるようにする。開発に必要なツール（StripeアカウントやCLI）も準備します。

具体タスク:

1. **Flaskプロジェクトの作成:** 新しいディレクトリを作り、仮想環境を有効化（任意）してFlaskをインストール。シンプルな `app.py` を作成し、Hello Worldを返すエンドポイントを実装します。
2. **Dockerファイルの用意:** PythonベースのDockerイメージを利用し、作成したFlaskアプリをコンテナ化するための `Dockerfile` を作成します。
3. **環境変数の管理:** StripeのAPIキーなど機密情報は環境変数で管理します（この時点では仮のプレースホルダを用意し、本番では実際の値をセットします）。
4. **Dockerビルドと実行:** Dockerイメージをビルドし、コンテナを起動。Flaskアプリをコンテナ内で動かし、ホストのブラウザから動作確認できるようにポートマッピングします（例: `docker run -p 5000:5000 ...`）。
5. **Stripeの準備:** Stripeのアカウントをまだ持っていなければ作成し、Stripe CLIもインストールしておきます（後のWebhookテストで使用します）。

コード例（Flask）: 初期段階では、Flaskアプリが正常に動作することを確認するために簡単なエンドポイントを用意します。

```
# app.py (Day0用シンプル版)
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello, Flask!"  # シンプルな動作確認用メッセージ

if __name__ == "__main__":
    # コンテナ内で実行する際、ホストからアクセス可能にするため0.0.0.0で待機
    app.run(host="0.0.0.0", port=5000)
```

上記コードをベースに、以降のDayで機能追加していきます。

コード例（Dockerfile）：同ディレクトリに次の内容で`Dockerfile`を配置します。これによりFlaskアプリをコンテナ化し、依存関係をインストールします。

```
# ベースイメージとして公式Pythonイメージを使用
FROM python:3.10-slim

# 作業ディレクトリを設定
WORKDIR /app

# アプリの依存関係をコピー＆インストール（必要に応じてrequirements.txtを使用）
COPY requirements.txt ./
RUN pip install -r requirements.txt

# アプリケーションのソースコードをコピー
COPY . .

# 環境変数（Stripeの公開鍵・秘密鍵など）を渡すことを想定
# 本番では `docker run -e` オプションやdocker-composeで注入

# Flaskアプリを起動（本番ではGunicorn等を使用することも推奨）
CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

上記では簡易のためFlask開発サーバをそのまま起動しています。開発中はこれで十分ですが、必要に応じて`FLASK_APP=app.py`や`FLASK_ENV=development`等の環境変数をDocker実行時に指定してください。

Stripe側の設定や操作: - Stripeのアカウントを作成してダッシュボードにログインします。テストモードに切り替え、**APIキー**（公開可能キーとシークレットキー）を取得します。Stripeのキーは**テスト用**と**本番用**がそれぞれ公開鍵・秘密鍵のペアで計4つ発行されています [1]。開発時はテスト用のキーを使用し、秘密鍵は外部に漏らさないよう注意してください [1]。取得したテスト用の秘密鍵・公開鍵は、この後の実装で環境変数`STRIPE_SECRET_KEY`および`STRIPE_PUBLISHABLE_KEY`として設定します。 - Stripe CLIのセットアップ: 開発環境にStripe CLIをインストールし、`stripe login`コマンドで自分のStripeアカウントに認証しておきます（成功するとCLIから自動でブラウザが開き認可されます）。

確認方法と通過チェックリスト:

- Dockerコンテナをビルドし起動: `docker build -t flask-stripe .` → `docker run -p 5000:5000 flask-stripe`。コンテナ内でFlaskアプリが立ち上がること。
- ブラウザで `http://localhost:5000` にアクセスし、「Hello, Flask!」のメッセージが表示されること。
- ターミナルのログにFlaskの起動メッセージが出力され、アクセス時にコンソールログが記録されていること（標準出力がDocker経由で見えるはずです）。
- Stripeダッシュボードにアクセスでき、テスト用APIキーの取得が完了していること。

注意点・トラブルシュート:

- ポート関連: Dockerで起動する際に `-p 5000:5000` でポートフォワードを設定しないとホストからアクセスできません。また、Flaskアプリを `0.0.0.0` で待ち受けさせないとコンテナ外部からのアクセスがブロックされます。この設定漏れによって「サイトに接続できない」場合は、Dockerfileや起動コマンドのホスト/ポート設定を確認してください。
- Dockerビルドのキャッシュ: `requirements.txt` を変更したのにパッケージが更新されない場合、一度Dockerイメージのキャッシュを破棄するか、`docker build --no-cache` オプションを使って再ビルドしてください。
- 環境変数の取り込み: Dockerで環境変数を渡す方法はいくつかあります。開発中はホストのシェルから `-e STRIPE_SECRET_KEY=sk_test_xxx` のように渡すか、`.env` ファイルを用意してDocker Composeで読み込む方法があります。秘密情報をDockerfile内に直書きしないようにしましょう（Dockerイメージに埋め込むと漏洩のリスクがあります）。

次フェーズに進む前に満たすべき条件:

- Flaskアプリの基本構造ができ、`/` エンドポイントで簡単なレスポンスを返せている。
- Dockerコンテナ上でアプリが正しく起動し、ホストからのリクエストを受け付けられている。
- StripeのテストAPIキー（公開鍵・秘密鍵）が入手できており、今後環境変数として利用できる準備が整っている。

Day1: Stripe Checkoutの導入と決済セッションの開始

ゴール: FlaskアプリからStripeのCheckoutセッションを作成し、ユーザがStripeの決済画面で支払いを行えるようにする。テスト用のカードで実際に決済を試し、Stripe上で取引が生成されることを確認します。

具体タスク:

1. **Stripeライブラリの導入:** FlaskアプリにStripeの公式Python SDK (`stripe` モジュール) を追加します（`pip install stripe`）。これをDocker環境にも反映させるため、requirements.txtに `stripe` を追記しDockerイメージを再ビルドします。
2. **APIキーの設定:** Day0で取得したStripeのテスト用APIキー(公開鍵・秘密鍵)をFlaskアプリで使用できるようにします。例えば環境変数から読み込んで、アプリ起動時に `stripe.api_key` に秘密鍵をセットします。
3. **Checkout Session作成エンドポイントの実装:** Stripe Checkout用のセッションを生成するエンドポイント（例: `/checkout`）をFlaskに追加します。このエンドポイントでStripeの `checkout.Session.create(...)` を呼び出し、成功時に返される `session.url`（Stripeホスト上のCheckout画面URL）へリダイレクトするようにします。

4. セッション生成には、成功時リダイレクト先`success_url`とキャンセル時リダイレクト先`cancel_url`を指定します。これらは自分のアプリ内のURL(例: `http://localhost:5000/success`等)を指定します。
5. 決済の内容を`line_items`として渡します。ここでは簡単のため、ハードコードで商品名と価格を指定します（Stripeに事前登録した価格IDを使う場合は`price`を指定できますが、ここではコード内で金額等を指定します）。
6. 決済モード`mode="payment"`（単発支払い）や支払い方法`payment_method_types=["card"]`も指定します。
7. **成功/キャンセル用のルート用意:** Checkout後にStripeからリダイレクトされる先となる`/success`および`/cancel`エンドポイントをFlaskに実装します。初期段階では、それぞれ「支払いに成功しました」「支払いをキャンセルしました」等の簡単なメッセージを表示するだけでOKです。
8. **UIのトリガー準備:** ユーザが支払いを開始できるように、アプリのトップページ（`/`）に決済開始用のリンクやボタンを配置します。シンプルに`<a href="/checkout">支払う</a>`とするか、フォームボタンでも構いません。

コード例（Flask）： Stripe Checkoutセッションを生成し、Stripeのホストする決済ページへリダイレクトする実装例です（Day0の`app.py`に追記・修正）。

```
import os
import stripe
from flask import Flask, redirect

app = Flask(__name__)

# 環境変数からStripeの公開鍵・秘密鍵を取得
stripe.api_key = os.getenv("STRIPE_SECRET_KEY", default="")  # テスト秘密鍵をセット
PUBLIC_KEY = os.getenv("STRIPE_PUBLISHABLE_KEY", default="") # テスト公開鍵（必要ならフロントで使用）

@app.route("/")
def index():
    # シンプルなホーム。実際はテンプレートを使ってボタンを配置可能
    return '<h1>デモショップ</h1><p><a href="/checkout">テスト商品を購入</a></p>'

@app.route("/checkout")
def checkout():
    # ドメインURLを組み立て（Dockerの場合ホスト名に注意。ローカルテスト用にlocalhost使用）
    base_url = "http://localhost:5000/"
    try:
        # StripeのCheckout Sessionを作成
        checkout_session = stripe.checkout.Session.create(
            success_url = base_url + "success",     # 支払い成功後に戻るURL
            cancel_url  = base_url + "cancel",      # 支払いキャンセル時に戻るURL
            payment_method_types = ["card"],        # 使用する支払方法
            mode = "payment",                       # 単発の支払いモード
            line_items = [                          # 商品ラインアイテムの指定
                {
                    'price_data': {
                        'currency': 'jpy',
                        'product_data': {'name': 'テスト商品'},
                        'unit_amount': 5000,  # 単価50円（通貨の最小単位：50円を100倍の
```

```
50*100=5000で指定）
                        },
                        'quantity': 1,
                    }
                ]
            )
        except Exception as e:
            return f"Error creating checkout session: {e}", 500

        # Stripeの決済ページURLへリダイレクト
        return redirect(checkout_session.url, code=303)

@app.route("/success")
def success():
    return "支払いが完了しました！ありがとうございます。"

@app.route("/cancel")
def cancel():
    return "支払いがキャンセルされました。"
```

上記コードでは、`/checkout`エンドポイントにアクセスするとStripeのチェックアウトページにリダイレクトします。ユーザはそこでカード情報を入力して支払いを完了し、結果に応じて`/success`または`/cancel`に戻ってきます。

Stripe側の設定や操作（ダッシュボード含む）：

- **製品と価格の設定（必要に応じて）：**コード内で商品名と価格を指定しましたが、Stripeダッシュボード上で商品(Product)と価格(Price)を作成しておき、`Session.create`時に`line_items=[ {'price': 'price_XXXXXXXX', 'quantity': 1} ]`のように設定することもできます。こちらの方法ではコード内に金額をハードコーディングせずに済みます。今回は手軽さを優先しコード上で指定しましたが、ダッシュボードで**テストデータ**として商品を作成してみるのも良いでしょう。
- **テスト用カード情報:** Stripeはテストモードで使用可能なカード番号を公開しています [2]。例えば「4242 4242 4242 4242」は常に承認されるテストカード番号です [2]。有効期限は未来の日付、CVCは任意の3桁、郵便番号も任意の数字で構いません [2]。ダッシュボードの「開発者用」セクションでこれらテストカードの一覧を確認できます。
- **APIキーの確認:** Flaskアプリ側で環境変数に設定した公開鍵・秘密鍵が正しいテスト用キーであることを再度確認します。万一間違って本番キーを使うと、テストモードで意図した動作が行えません。

確認方法と通過チェックリスト:

- ブラウザでアプリのトップページ（`http://localhost:5000/`）にアクセスし、「テスト商品を購入」のリンクをクリックするとStripeのチェックアウト画面にリダイレクトされること。
- Stripe Checkout画面で上記のテストカード番号（例: `4242 4242 4242 4242`）と適当な有効期限・CVCを入力し、決済を試行します。
- **成功ケース:** 情報を正しく入力すると決済が即時に承認され、数秒後に自動で`http://localhost:5000/success`ページに遷移するはずです。ページに「支払いが完了しました！」と表示されれば成功です。
- **キャンセルケース:** カード情報入力画面で「キャンセル」やブラウザ戻る操作をした場合、`http://localhost:5000/cancel`に飛び「支払いがキャンセルされました。」と表示されます。
- Stripeダッシュボードで決済の記録を確認します。ダッシュボードの「支払い(Payments)」メニューを開くと、今実施した取引がテストデータとして記録されているはずです [3]。取引の金額やステー

タス（Succeededなど）が正しいことを確認してください。また「イベント(Events)」欄に `checkout.session.completed` というイベントが発生していることも確認できます（このイベントは後ほどWebhookで利用します）。
- ターミナルのFlaskログでも、`/success` や `/cancel` ページにアクセスされた記録が残ります。これによりStripeからユーザが戻ってきたことが裏付けられます。

注意点・トラブルシュート:

- 決済画面にリダイレクトしない: リダイレクト先URLが無効な場合（例: ベースURLを誤って `localhost` ではなく別ホストにしている等）、Stripe側で「redirect_url_invalid」のエラーが発生し決済画面が開きません。`success_url` や `cancel_url` に正しい自分のアプリのアドレスを設定しているか確認してください。
- セッション作成時のエラー: `stripe.checkout.Session.create` で例外が発生する場合、APIキーが正しく設定されていない可能性があります。また、Dockerコンテナ内でインターネットにアクセスできない環境だとStripe APIに接続できません（通常ローカル開発環境なら問題ありません）。エラーメッセージをキャッチして出力するようにしてあるので、ログに表示された内容をもとに原因を特定してください。
- 価格の単位: Stripeでは通貨ごとに最小単位で金額を指定します。日本円(JPY)の場合は1円=100 (最小通貨単位: セン)。コード例では50円を `unit_amount=5000` と指定していますが、仮に5000と書けば50円になる点に注意してください。混乱する場合は、Stripeダッシュボードで Priceを作成し、そのIDを指定する方法を取るとミスが減ります。
- 公開鍵の利用: この段階ではシンプル化のためサーバー側から即リダイレクトしていますが、通常はStripeの公開可能キーをフロントエンド（ブラウザ）でStripe.jsに渡し、`stripe.redirectToCheckout({ sessionId })` を呼ぶ実装もあります [4] 。今回は説明を簡単にするためバックエンドから直接リダイレクトしていますが、どちらの方法でも構いません。

次フェーズに進む前に満たすべき条件:

- テスト用カードを使い、正常にStripe上で決済が完了できる（Stripeのチェックアウトページに遷移し、成功後にアプリに戻ってこられる）。
- 成功時とキャンセル時の挙動がそれぞれ `/success`・`/cancel` で確認できる。
- Stripeダッシュボード上で、決済が記録されイベント `checkout.session.completed` が発生している。
- ここまではあくまでユーザ画面上の動作確認です。次フェーズではサーバー側で支払い結果を自動処理するWebhook機能を実装します。進む前に、上記の手順で手動テストが問題なく行えることをチェックしてください。

Day2: 決済フローの確認とStripeダッシュボードの活用

ゴール: Stripe Checkoutを用いた基本的な決済フローを一通り確認し、ユーザエクスペリエンスと開発者視点の両面で理解を深めます。成功・キャンセルページの内容を整え、Stripeダッシュボードで取引履歴やイベントログを確認する方法を習得します。

具体タスク:

1. **成功/キャンセルページの整備:** Day1で実装した `/success` および `/cancel` ルートの出力を必要に応じて改善します。簡単なHTMLテンプレートを用いてユーザにメッセージを表示したり、決済IDや購入内容を表示することもできます（後者はWebhook導入後に実装可能になります）。

2. **キャンセル処理のテスト:** Stripeチェックアウト画面で「キャンセル」を選んだ場合の動きを試し、`/cancel`ページに戻ることを確認します。ユーザが支払いを途中で取りやめたケースでもアプリが問題なく動作することを確認しましょう。
3. **Stripeダッシュボードでの検証:** 開発者コンソール（ダッシュボードの「開発者」セクション）で以下を確認します。
4. **Payments（支払い）履歴:** テストモードに切り替え、一覧に自分の取引が表示されていることを確認します。決済額、手数料（テストでは¥0）、支払い方法（カード種類）などの詳細を確認できます。
5. **イベントログ:** 「イベント」メニューでは、Webhook送信対象となる各イベントの履歴が見られます。`checkout.session.completed`イベントを選択すると、そのペイロード詳細を確認できます。どのようなデータ（セッションID、金額、通貨、顧客メール等）が含まれているか見てみましょう。これは後でWebhookハンドラで処理するデータと同じものです。
6. **UI/UXの見直し:** 必要に応じ、トップページに決済ボタンを設置するなどUIを改善します。まだ簡易なテキストリンクだけでも動作上は問題ありませんが、将来的にボタン操作にする場合はJavaScript経由でCheckoutセッションを開始する実装（Stripe.jsの利用）も検討できます。
7. **コードの整理:** Day1までで追加・変更したコード（特に`app.py`）をリファクタリングします。マジックナンバー（通貨単位や商品名など）を定数にしたり、公開鍵をフロント用に渡す必要があればエンドポイントを用意する（例えば`/config`で`{"publicKey": PUB_KEY}`を返す）など、学習ついでにコードを読みやすくしておきます。

Stripe側の設定や操作:

- **Stripeダッシュボードの活用:** 開発中はStripeダッシュボードを頻繁に確認する習慣を付けましょう。Payments欄では決済履歴が一覧でき、特定の取引をクリックするとその詳細（セッションIDや関連するPaymentIntent ID、顧客情報等）が表示されます。Events欄では各イベントとWebhook配信状況が確認できます。現時点ではWebhookエンドポイントを登録していないため「送信済みエンドポイント」は0件になっていますが、次フェーズでエンドポイントを追加するとここに配信ログが表示されるようになります。
- **テストデータモード:** ダッシュボード右上の「テストデータ」トグルが有効になっていることを確認してください。テストモードと本番モードでデータが分離されているため、テスト中に本番モードにするとデータが見当たらず混乱します。常にオレンジ色の「テストデータ」表示が出ている状態で作業します。

確認方法と通過チェックリスト:

- Stripe Checkoutの成功・キャンセルの両パターンを手動で再度テストし、期待通りそれぞれのページに遷移すること。
- Stripeダッシュボード上で、該当の支払いが**テストモード**で記録されていることを確認。金額・日時・ステータスなど正確か。
- イベントログを開き、`checkout.session.completed`イベントの中身を閲覧できること。ここで表示されるJSONデータに含まれる`id`（evt_xxx…）や`type`、`data.object.id`（checkout sessionのID）、`amount_total`などを把握します。次のWebhook処理でこれらデータを扱います。
- （オプション）Stripeダッシュボードからメール送信設定やレシート設定を確認します。テストでは実際の決済通知メール等は送られませんが、本番モードではStripeに顧客メールアドレスが渡っていればレシートメールを送信する機能もあります。

注意点・トラブルシュート:

- 重複テスト: 同じ決済フローを何度試しても問題ありませんが、Stripeダッシュボード上でデータが蓄積していきます。必要であれば「テストデータのリセット」機能を使ってリセット可能です（**注意:** テストとはいえ履歴が消えるので安易なリセットは推奨されませんが、Sandbox環境として割り切るならOKです）。

- ダッシュボードの時間表示: Stripeのイベントログのタイムスタンプは UTC基準だったりブラウザ設定によって表示タイムゾーンが異なる場合があります。自分の操作時刻と見比べてずれていても慌てないようにしましょう。
- イベントの重複: 現時点では `checkout.session.completed` のみ確認していますが、実際には一度の決済で複数のイベント（例えば `payment_intent.succeeded` や `charge.succeeded` 等）も発生します。今回のカリキュラムでは主にCheckout Sessionベースで処理しますが、Stripeでは様々なイベントが飛んでくることを頭に入れておいてください。
- セキュリティ: フロントエンド側でStripeの公開可能キーを使ってCheckoutへリダイレクトする実装を行う場合、公開鍵を直接HTMLやJSに埋め込みます。その際、公開鍵は漏れても安全な情報ですが、誤って秘密鍵を埋め込まないよう注意しましょう。

次フェーズに進む前に満たすべき条件:

- 基本的な決済フローに一通り慣れ、Stripe上に記録されたデータを確認できている。
- アプリ側のページ遷移や表示に不備がなく、ユーザが操作する一連の流れがスムーズである。
- Stripeダッシュボード上でイベント内容を把握したことで、「このデータを使って自分のサーバー側を更新すれば良い」というイメージが持てている。
- これでフロントエンド主体の確認は完了です。次は、**サーバー側で支払い完了を検知して処理するWebhook**の実装に移ります。

Day3: Stripe Webhookエンドポイントの実装とテスト

ゴール: Stripeからのイベント通知（Webhook）を受け取るためのエンドポイントをFlaskアプリに追加し、Checkout完了イベントをサーバーサイドで処理できるようにする。Stripe CLIを用いてローカル環境でWebhook受信をテストします。

具体タスク:

1. **Webhookエンドポイントの追加:** FlaskアプリにWebhook受信用のルート（例: `/webhook`）を作成します。メソッドは `POST` のみ許可し、StripeからのHTTPリクエストを受信できるようにします。
2. **リクエスト検証とパース:** 受信したリクエストのボディ（JSONペイロード）と、Stripeが付与する署名ヘッダー（`Stripe-Signature`）を用いて、正当なイベント通知かを検証します。Stripeの提供する `stripe.Webhook.construct_event(payload, sig_header, endpoint_secret)` メソッドを使用します[5]。このとき必要となる `endpoint_secret`（Webhook署名検証用シークレットキー）は、Stripe CLIまたはStripeダッシュボードで取得します。
3. **Stripe CLIでのセットアップ:** ターミナルで `stripe listen --forward-to localhost:5000/webhook` を実行します。これによりStripeアカウント上のイベントが発生した際、ローカルの `/webhook` エンドポイントに中継されます。また実行直後に「Webhook署名シークレット」（`whsec_...` 形式のキー）が表示されます[6]。このキーを環境変数（例えば `STRIPE_WEBHOOK_SECRET`）に保存し、Flaskアプリ内で `endpoint_secret` として使用します。
4. **署名の検証:** `construct_event` を呼ぶ際、`payload` は**生のリクエストボディ**（`request.data`）を渡します[7]。また `sig_header` には `request.headers['Stripe-Signature']` の値を渡します[7]。これらと先ほどの `endpoint_secret` を合わせ、Stripe発行のイベントであることを検証します。検証に失敗した場合はただちにエラーを返します（HTTP 400）[8]。検証成功すれば `event` オブジェクトが得られるので、次のステップへ進みます。
5. **イベントタイプの判定:** 取得した `event` オブジェクトの `type` 属性を確認し、`"checkout.session.completed"` の場合に処理を行います。Day2で確認したように、このイベントは支払い完了時に送信されます。現段階では、処理としてログにメッセージを出力する程度に留めます（後で台帳に記録する処理を追加します）。例えば、`session = event['data']['object']` から

Checkout Sessionの詳細を取り出し、`session['id']`や`session['amount_total']`を表示するなどします。

6. **素早いレスポンスの返却:** Webhookの受信処理では、**できるだけ早くHTTP 200レスポンスを返す**ことが重要です [9]。Stripe側は受信先が200台ステータスを返すまで何度もリトライするため、重い処理は後回しにし、まずは`return "", 200`で確実に応答しましょう [9]。今回の実装でも、ログ出力後すぐに`200 OK`を返すようにします。
7. **ローカルでのWebhookテスト:** Stripe CLIを使ってWebhookエンドポイントが正しく動作するか検証します。
8. 別のターミナルで`stripe listen --forward-to localhost:5000/webhook`を起動しておく（上記2で設定済み）。
9. 新しいターミナルから`stripe trigger checkout.session.completed`を実行します。これはStripe CLIがテスト用に用意したイベント生成コマンドで、`checkout.session.completed`イベントを疑似的に発生させてくれます。実行すると、CLI側にイベントが送られた旨と、我々のFlaskエンドポイントにリクエストを転送したログが表示されます。
10. Flaskアプリのログを確認し、想定したメッセージ（例えば「Webhook受信: session_complete (ID: ...)」等）が出力されていれば成功です。ステータス200も返っているため、CLI側には「Webhook received successfully」のような表示がでます。

コード例（Flask）: Webhookエンドポイントの骨組み実装例です（Day1/Day2の`app.py`に追記）。

```
from flask import request, jsonify

# （前略）上部は省略。Day1までのコードに続けて追記します。

# 環境変数からWebhook署名検証用のシークレットを取得
WEBHOOK_SECRET = os.getenv("STRIPE_WEBHOOK_SECRET", default="")

@app.route("/webhook", methods=["POST"])
def stripe_webhook():
    # Webhookイベントを受け取るエンドポイント
    payload = request.data  # 生のリクエストボディ
    sig_header = request.headers.get("Stripe-Signature", "")
    event = None
    try:
        # 署名の検証とイベントオブジェクトの取得
        event = stripe.Webhook.construct_event(payload, sig_header, WEBHOOK_SECRET)
    except stripe.error.SignatureVerificationError as e:
        # 署名検証エラー（偽装やシークレット不一致）
        print("⚠ Webhook signature verification failed.")
        return jsonify({'error': 'invalid signature'}), 400
    except Exception as e:
        # その他エラー（ペイロード不正など）
        print(f"⚠ Webhook error: {e}")
        return jsonify({'error': 'webhook error'}), 400

    # イベントタイプによって処理を分岐
    if event["type"] == "checkout.session.completed":
        session = event["data"]["object"]
        session_id = session.get("id")
        amount_total = session.get("amount_total")  # 合計金額 (支払額)
        currency = session.get("currency")
```

```
        print(f"   Payment succeeded for Session {session_id}: {amount_total}{currency}")
        # （この後のDay4で台帳記録を実装）
    else:
        # 上記以外のイベントはひとまずログに出すだけ
        print(f"Received event type: {event['type']}")

    # 素早く成功レスポンスを返す
    return "", 200
```

この実装では、`checkout.session.completed`イベントを受信した際に、セッションIDと金額をログ表示しています（実際の処理はDay4で追加）。また、署名検証に失敗した場合や想定外のエラーが起きた場合には400エラーで応答します。正常時は必ず200を返すことで、Stripeに「受信成功」を伝えています [9]。

Stripe側の設定や操作:

- **Webhookエンドポイントの登録（開発時は不要）:** Stripeダッシュボードの「Webhook」設定画面では、本来ここでエンドポイントURLを登録しどのイベントを受信するか設定します [10]。ただしローカル開発中は外部から直接呼び出せないため、Stripe CLIを使ってトンネリングしています。したがって**ダッシュボード上でのWebhook設定はまだ行う必要はありません**（むしろ登録してもローカルには届かないので意味がない）。本番運用時に改めて設定します。
- **Stripe CLIでのテスト:** `stripe listen`と`stripe trigger`コマンドを適切に使うことで、ダッシュボードを経由せずともイベント送信テストが可能です [6]。特に`stripe trigger`は主要なイベントをシミュレートできます。CLIを使うことで決済を毎回手動実行せず素早くイベント処理を確認できるので活用しましょう。
- **イベント内容の確認:** CLI経由で受信したイベントは、Stripe CLIの出力に詳細が表示されます。成功した場合、Stripe CLIのコンソールに受信したJSONペイロードがそのまま表示されます。それを見れば、自分がパースしている`event["data"]["object"]`の中身（先ほどダッシュボードで見たJSONと同様）を再確認できます。

確認方法と通過チェックリスト:

- `stripe listen --forward-to localhost:5000/webhook`を起動したターミナルに「Ready!（接続待機中）」と表示されていることを確認。
- 別ターミナルから`stripe trigger checkout.session.completed`を実行し、以下をチェック:
- Stripe CLI側に「✔ Webhook received by localhost:5000/webhook [200]」等と表示されること（受信が成功し200応答を返した印）。
- Flaskアプリのログに、先ほど実装した`print`によるメッセージ（セッションIDと金額を含む行）が出力されていること。
- Stripe CLI側にイベントのJSONペイロードが表示され、`type`が`checkout.session.completed`であること、および`status: succeeded`や`payment_status": "paid"`のような情報が含まれていること。
- 上記が確認できたら、今度は実際の決済を発生させてWebhookをテストします。先ほどと同様に`stripe listen`を起動したまま、ブラウザでアプリから決済を行ってみます。支払い完了後、Stripe CLIが`checkout.session.completed`イベントを検知して転送し、同様にログが出力されればOKです。
- 署名検証を有効にした状態でテストできたため、セキュリティ上も正しいWebhook受信となっています。Stripe CLIが表示した`Webhook signing secret`は開発用（テスト用）ですので、誤ってGit等に公開しない限り問題ありません。本番環境では別途ダッシュボードから取得することになります。

注意点・トラブルシュート:

- 署名検証エラー: Stripe CLIの `listen` 開始時のシークレットと、コード内の `WEBHOOK_SECRET` が一致していないと `SignatureVerificationError` が発生します [11]。例えば、Webhookエンドポイントを作り直したりCLIを再起動するとシークレットが変わるので、その際は環境変数も更新してください。「No signatures found matching the expected signature」というエラーが出た場合、まずシークレット値の取り違え（Webhook **ID**ではなく**シークレット**を使う必要があります）を疑ってください [11]。
- ローカル環境での公開URL: Stripe CLIを使わずにWebhookテストをしたい場合、ngrokなどでローカルサーバーを一時的に公開URLにマッピングする方法もあります。しかしStripe CLIが推奨されており簡便ですので、本手順ではCLIを用いました [6]。
- マルチイベント: 現状の実装では特定の `checkout.session.completed` しか処理していません。他のイベントが来ると一律で「Unhandled event」的なログを出すだけです。開発中はこれで十分ですが、実際には不要なイベントはStripeダッシュボード側で購読しない設定にするか、コード側で無視するかを検討します。
- レスポンス速度: Webhookの受信処理内で外部システムとの通信や長時間処理を行うと、Stripeはタイムアウトと判断してWebhookを再送してきます [9]。その結果、同じイベントが複数回届き重複処理の原因になります。必ず**即時に202もしくは200で応答を返す**ように心掛け、実際の処理は後段（非同期処理やキューイングなど）に回す設計が推奨されています [9]。
- デバッグ: Stripe CLIは受信したWebhookリクエストをそのまま出力してくれるため、開発時のデバッグに非常に便利です。署名検証を一時的に無効化してペイロードだけ受け取りたい場合、`stripe.Webhook.construct_event` の部分をコメントアウトし、`event = json.loads(request.data)` とすることで署名なしでもテストできます。ただし検証を外すと不正リクエストを受けても気付けないので、一時的なデバッグ用途に留めます。

次フェーズに進む前に満たすべき条件:

- Stripe CLI経由で `checkout.session.completed` イベントを受け取り、サーバー側でそれを検知・ログ出力できている。
- 実際の決済完了時にもWebhookイベントを受け取れている（ローカルで支払いテストを行い確認）。
- Webhookの署名検証が正しく行われ、不正なリクエストを拒否できる状態になっている。
- これで「支払い完了をサーバーが知る」基盤ができました。次は、この受け取った情報を**台帳（Ledger）として保存**し、アプリ内で取引履歴を管理します。

Day4: 支払い台帳の実装とデータベースへの記録

ゴール: Webhookで受け取った支払い完了情報を自社システム内に**台帳**として蓄積する。具体的には、簡易的なデータベース（SQLite）に取引情報を保存し、後から参照できるようにする。これにより、Stripeの情報と自前の記録を付き合わせ（突合）できるようになります。

具体タスク:

1. **データベースのセットアップ:** シンプルなSQLiteデータベースを使用します。Dockerコンテナ内で永続化するため、SQLiteのDBファイル（例: `ledger.db`）を作成し、コンテナのファイルシステムに保存します（必要に応じてDockerボリュームをマウントしてホストに保存しても構いません）。
2. **テーブル定義:** 台帳として保持したい情報を決め、データベースにテーブルを作成します。最低限、StripeのCheckout Session ID、金額、通貨、支払いステータス、日時を保存するとよいでしょう。例えば以下のようなテーブルを想定します:
3. `ledger` テーブル:
    - `session_id` (テキスト型, 主キー) – StripeのCheckout SessionのID（ユニーク）

    - `amount` (整数型) – 支払い金額（最小通貨単位, 例: 5000は¥50を表す）
    - `currency` (テキスト型) – 通貨（例: "jpy"）
    - `status` (テキスト型) – ステータス（例: "paid" 支払完了, "canceled" キャンセル等）
    - `created` (タイムスタンプ型) – レコード作成日時（イベント受信日時）
4. **テーブルの作成:** アプリ起動時に上記テーブルが存在しなければ作成する処理を入れます。SQLiteは手軽に使えるので、Pythonのsqlite3モジュールで接続し、`CREATE TABLE IF NOT EXISTS` クエリを実行します。この処理は `app.py` 内でグローバルに実行しておいても良いですし、初回Webhook受信時に行っても良いでしょう。ここでは確実にするためアプリ開始時に実行しておきます。
5. **Webhookハンドラの拡張:** Day3で実装した `/webhook` エンドポイント内の処理を修正し、`checkout.session.completed` イベントを受け取った際に上記データベースへレコードを挿入します。具体的には、`session = event["data"]["object"]` から各種情報を取得し、`INSERT` クエリを実行します。
6. 重複挿入を避けるため、`session_id` をプライマリキーに設定しておけば、同じSession IDで二重登録しようとした際にエラーになります（Stripeの再送により同一イベントが届く可能性があるため、これで二重計上を防ぎます [12]）。
7. データベース操作に時間をかけすぎないよう注意します。実際のところINSERT程度なら一瞬ですが、万一ここでボトルネックになる場合は処理をキューに投げる検討も必要です。今回はシンプルに同期処理します。
8. **台帳データの確認手段:** 台帳に記録された内容を確認できるようにします。簡易的な方法として、現在の台帳内容を表示するルート（例: `/ledger`）を作成し、テーブル内の全レコードを取得してJSONやHTMLで返します。開発用途のエンドポイントですが、動作検証に役立ちます。
9. **テスト実施:** Stripe CLIを使って複数回イベントを発生させ、また実際にブラウザから複数回決済を行い、台帳に期待通りレコードが追加されるか確認します。重複がないこと、金額や通貨が正しく記録されていることをチェックします。

コード例（Flask）: Webhook処理に台帳記録を組み込んだ例です。

```
import sqlite3
from datetime import datetime

# （前略）既存のコードに追加・修正します

# アプリ起動時にSQLite接続しテーブルを作成
conn = sqlite3.connect("ledger.db", check_same_thread=False)
cursor = conn.cursor()
cursor.execute(
    "CREATE TABLE IF NOT EXISTS ledger ("
    "session_id TEXT PRIMARY KEY, "
    "amount INTEGER, "
    "currency TEXT, "
    "status TEXT, "
    "created TEXT)"
)
conn.commit()

@app.route("/webhook", methods=["POST"])
def stripe_webhook():
    payload = request.data
    sig_header = request.headers.get("Stripe-Signature", "")
    try:
        event = stripe.Webhook.construct_event(payload, sig_header, WEBHOOK_SECRET)
```

```
    except stripe.error.SignatureVerificationError:
        return "Signature verification failed", 400
    except Exception as e:
        return f"Webhook error: {e}", 400

    if event["type"] == "checkout.session.completed":
        session = event["data"]["object"]
        session_id = session.get("id")
        amount_total = session.get("amount_total", 0)
        currency = session.get("currency", "")
        payment_status = session.get("payment_status", "")  # "paid" 等
        # 台帳に挿入
        try:
            cursor.execute(
                "INSERT INTO ledger (session_id, amount, currency, status, created) VALUES
(?, ?, ?, ?, ?)",
                (session_id, amount_total, currency, payment_status,
datetime.utcnow().isoformat())
            )
            conn.commit()
            print(f"  Ledger updated: Session {session_id} recorded.")
        except sqlite3.IntegrityError:
            # 重複の可能性（既に登録済みのsession_id）
            print(f"⚠ Session {session_id} is already recorded in ledger.")
    return "", 200

@app.route("/ledger", methods=["GET"])
def show_ledger():
    cursor.execute("SELECT session_id, amount, currency, status, created FROM ledger")
    rows = cursor.fetchall()
    # シンプルにテキストで一覧表示（実際は適切にフォーマットする）
    result = "SessionID, Amount, Currency, Status, Created\n"
    for r in rows:
        result += f"{r[0]}, {r[1]}, {r[2]}, {r[3]}, {r[4]}\n"
    return "<pre>" + result + "</pre>"
```

Stripe側の設定や操作:

- 特に新規のダッシュボード操作は必要ありません。引き続きStripe CLI経由でWebhookイベントを送ってテストします。
- Stripeダッシュボードで**イベント履歴**を確認すると、これまでテストで発生させた `checkout.session.completed` イベントが一覧できます。各イベントを開くと「配信の試行」という欄があり、Stripe CLI経由の場合「（エンドポイント名）Stripe CLIによって配信」と表示されます。正式にWebhookエンドポイントを登録すればここにエンドポイントURLが表示され配信状況が記録されますが、開発中はCLIに配信されているためそのように表示されます。

確認方法と通過チェックリスト:

- `stripe trigger checkout.session.completed` を2回以上実行し、それぞれについてアプリのログに「Ledger updated: Session ... recorded.」のメッセージが出力されること。

- ブラウザで `http://localhost:5000/ledger` にアクセスし、Ledgerに記録された内容が一覧表示されていること。実行回数分のレコードが増えているか確認します。
- Ledger表示の各項目（session_id, amount, currency, status, created）が適切に埋まっていること。特に `amount` と `currency` がStripeダッシュボード上の決済金額・通貨と一致するか確認します。日本円50円の場合、amountは「5000」、currencyは「jpy」と記録されているはずです。
- Stripeダッシュボードのイベントログと照らし合わせて、Ledger上のレコード数・内容が一致することを確認します。例えばイベントログに3件の `checkout.session.completed` があれば、Ledgerにも3件のレコードがあるかチェックします。**これが台帳とStripeの突合の第一歩です。**
- 重複処理防止のテスト: Stripe CLIで**同じイベントID**を再送信してみます。まず、CLIで直近のイベントIDを取得するか、ログからコピーします（イベントIDは `evt_` で始まる文字列）。次に `stripe events resend evt_xxx --forward-to localhost:5000/webhook` のように実行し、そのイベントを再送します。アプリのログに「already recorded」の警告が出て、Ledgerに二重登録されないことを確認します。

注意点・トラブルシュート:

- データベースロック: SQLiteはシングルコネクションでのシンプルな使用には問題ありませんが、別スレッドからアクセスするとロックがかかる場合があります。Flask開発サーバはシングルスレッドなので今回は大丈夫ですが、マルチスレッド環境では `check_same_thread=False` を指定するなど設定が必要です。
- 台帳のスキーマ拡張: 実際の用途に応じて、顧客IDや商品IDなども記録するとより実用的です。StripeのCheckout Sessionオブジェクトには `customer_details` や `metadata` など様々な情報が含まれるので、必要に応じてテーブル項目を増やしてください。
- 重複処理: コードではDBの主キー制約で重複挿入を防いでいますが、本来はアプリケーションレベルでも**冪等性**を考慮すべきです [12]。Stripeのドキュメントも「同じCheckout Sessionに対しては一度だけ処理するように」と注意喚起しています [12]。今回のケースではSession IDをキーに一度処理したら二度目以降スキップするロジックで十分でしょう。
- タイムスタンプ: 上記コードでは `datetime.utcnow().isoformat()` でUTC時刻を文字列保存しています。必要であればDATETIME型にしてタイムゾーン管理を行ってください。ログと比較する際はUTCで統一されていると照合しやすいです。
- Ledger参照の扱い: `/ledger` エンドポイントは開発・デバッグ用途です。本番では認証なしに取引情報を公開しないよう、必ず保護するか管理画面だけの機能にするなど対策が必要です。

次フェーズに進む前に満たすべき条件:

- Webhookで受け取った支払いデータが台帳データベースに保存できている。
- Stripe上のイベント/支払い履歴と、自前の台帳データが突合可能な状態になっている（少なくとも件数と金額が合っている）。
- 重複したWebhook受信（再送など）によって台帳が二重更新されない工夫が施されている。
- 以上により、「決済完了をトリガーに自社DBを更新する」という基本が達成できました。次は、複数回の決済テストや障害シナリオを試しつつ、ログやデータを確認して問題点を洗い出します。

Day5: 総合テストとトラブルシューティング

ゴール: ここまで構築したCheckout〜Webhook〜台帳の一連の仕組みを総合的にテストし、よくある落とし穴に対処できるようになる。Stripeのイベントログおよびアプリのログ・データベースを突き合わせ、想定外の事態が起きていないか確認する。問題があれば原因を特定し、次フェーズ移行前に解消します。

具体タスク:

1. **複数回の決済シナリオテスト:** テストカードを使って何度か支払いを行い、システムの挙動を確認します。様々な額や意図的にキャンセルも織り交ぜ、以下をチェックします。
2. 毎回StripeのCheckoutが正常に起動し、支払い結果に応じて正しいページに戻ってくること。
3. Webhookが毎回確実に受信・処理され、台帳に新規レコードが追加されること（キャンセルの場合、`checkout.session.completed`自体が発生しないため台帳追加はない想定です）。
4. Stripeダッシュボードの支払い履歴と台帳DBの内容が1対1で一致していること。例えば成功した支払いが5件なら、台帳にも5件の`paid`レコードが存在する。
5. **異常系テスト:** 可能な範囲でエラーパターンも試してみます。
6. Webhook署名シークレットを一時的に間違った値にしてイベントを送信し、署名検証エラーの挙動を確認（400を返し、Stripe CLIがリトライする様子をログで追う）。
7. （Stripe CLI機能を用いて）意図的にエンドポイントが500エラーを返すケースをシミュレートし、Stripe側で数分後に再送が行われることを確認。これは実際にはコードに一時的な不具合を仕込むなどして行います。重要なのは、Stripeは**4xx/5xxの応答時に複数回リトライする**という点です [13]。この動きにより重複イベントが届くことがあるため、既に対策した冗長性が有効であることを確認します。
8. 非同期決済方法のシミュレーション: カード以外の決済（例えば銀行振込等）をテストするのは難しいですが、Stripe CLIで`checkout.session.async_payment_succeeded`イベントを手動でトリガーしてみます（該当コマンドが用意されていれば）。本システムでは特に処理していませんが、ログに出るか確認します。
9. **ログ/データの監査:** Stripeダッシュボードのイベント配信ログ、アプリケーションのコンソールログ、および台帳データを総合的に見直し、不整合や見落としがないか確認します。具体的には:
10. Stripeダッシュボードの「Webhookの配信履歴」で失敗が1件でも記録されていないか（すべて200成功になっているか）。
11. アプリのログに予期せぬエラーメッセージや警告（⚠で始まる出力）がないか。
12. 台帳DB内に不自然な重複や欠損がないか（例えばsession_idが飛び番になっていないか、金額がおかしいものがないか等）。
13. **コードの最終調整:** 上記テストで発見した問題に対応します。例:
14. Webhookシークレットの扱いを環境変数からハードコーディングに一時変更していたら戻す。
15. ロギングをもう少し詳細にする（例えばどのIPからWebhook受信したか記録する等）ことも有用です。Flaskの`request.remote_addr`を使えばIPは取得できます。
16. 不要になったデバッグ用プリントを削除し、必要なログは`logging`モジュール経由で出すようリファクタリング（任意）。

Stripe側の設定や操作:

- **イベント種類の把握:** Stripeのドキュメントを参照すると、Checkout完了関連のイベントには`checkout.session.completed`の他に`checkout.session.async_payment_succeeded`や`checkout.session.async_payment_failed`があります [14]。今回は即時決済のみ扱いましたが、今後Bank Redirectなど非即時決済を扱う場合はこれらイベントも受信するようWebhookエンドポイントを拡張する必要があります [14]。
- **Webhook管理:** 開発中はStripe CLIに依存してきましたが、本番環境ではStripeダッシュボードにWebhookエンドポイントを登録します。**テストモード**と**本番モード**でエンドポイントは別々に設定可能です。テストで十分動作を確認したら、ダッシュボードの「エンドポイントを追加」からURL（例えば公開された本番サーバの`https://あなたのドメイン/webhook`）を登録し、受信するイベントタイプとして`checkout.session.completed`等を選択します [10]。登録後に表示される署名用シークレット値を本番環境の`STRIPE_WEBHOOK_SECRET`に設定するのを忘れないでください。
- **ダッシュボードでの再送:** Stripeはダッシュボード上からWebhookイベントの再送信が可能です。イベント詳細ページに「エンドポイントに再送信」ボタンが現れます。万一本番中にWebhook受信に失敗した場合、手動再送でリカバリできることを覚えておきましょう。

確認方法と通過チェックリスト:

- 複数回の決済とWebhook受信を行った結果、Stripeダッシュボードのイベント履歴と台帳DBのレコードが完全に一致している（件数・内容ともに）ことを確認。
- Stripe CLI/ダッシュボードのログで失敗や重試行の記録がないこと。仮に一度失敗させたテストをした場合も、最終的に成功扱いになるまでリトライされたかを確認。
- 予期しない例外やエラーがアプリログに出ていないこと。もし出ている場合、その原因を究明しコードを修正できたこと。
- システム全体として安定して動作していることをチーム内でデモし、他者から見ても要件（Checkout導入、Webhook処理、台帳記録）が満たされているか確認。

注意点・トラブルシュート:

- Sandboxでの限界: Stripeのテストモードは非常に本番に近い動きをしますが、唯一違うのは実際のお金が動かない点です。本番では銀行振込などでタイムラグのある支払いが発生しうるため、その点の考慮（上述のasyncイベント）を念頭に置いてください。
- ネットワーク障害: Webhook配信はStripe側で数日間リトライされますが、サーバが長期間ダウンしていると最終的に配信失敗に終わります。本番ではWebhookが届かなかった場合のバックアップ策（例えば管理者にアラートを送る、定期的にStripeのAPIから未処理のイベントをポーリングする等）も検討に値します [15] 。
- テストカードの種類: Stripeは各種シナリオをテストするカード番号を提供しています。例えば特定の番号で支払い失敗（カード拒否）を再現できます。興味があればStripeのテストカード一覧を参照し、`payment_intent.payment_failed`イベントなどが発生する状況も試すと理解が深まります。
- 並行処理: ごく短時間に多数の決済が発生した場合、Webhook受信が並行する可能性もあります。Flask開発サーバでは一度に1リクエストずつ処理しますが、Gunicornなどでワーカーを増やすと並行処理になります。そうなった場合でもデータ整合性（台帳の一意制約やトランザクション処理）が保たれるか、頭の片隅に置いておきましょう。
- ログレベル: 開発中は標準出力にプリントしていましたが、運用時は適切なログレベル（INFO, WARNING, ERRORなど）で記録するようにします。StripeのイベントIDやSession IDをログに入れておくと、後から問い合わせ対応する際に便利です。

次フェーズに進む前に満たすべき条件:

- Checkout〜Webhook〜Ledgerの一連の流れが安定して動作し、テストケースを網羅しても破綻しないことが確認できた。
- 予想されるエラーや障害に対して、適切な対策や監視ポイントが洗い出せている。
- 実装コードが整理され、次のステップ（本番移行や新機能追加）に進める状態になっている。
- これにてカリキュラム上の実装フェーズは完了です。最後のDay6では、成果物のドキュメント化と本番展開時の留意事項をまとめます。

Day6: READMEドキュメント整備と本番展開準備

ゴール:　ここまでの成果物をREADMEなどの**ドキュメント**にまとめ、誰が見ても手順が再現できるようにする。また、本番環境へのデプロイに向けて必要な設定の最終チェックを行います。

具体タスク:

1. **README.mdの作成:** プロジェクトの概要、セットアップ手順、使い方、環境変数の設定方法、動作確認方法などを網羅したREADMEを用意します。Day0〜Day5の内容を凝縮し、新たな開発者がREADMEを読めば環境構築から主要機能のテストまで出来るような構成が望ましいです。

2. プロジェクトの目的（Stripe Checkout＋Webhook＋台帳のデモ）
3. 使用技術（Flask, Docker, SQLite, Stripe API 等）
4. 環境構築手順（Dockerイメージのビルド/実行方法、必要な前提：Dockerインストール済み, Stripeアカウント必要 等）
5. 環境変数一覧（STRIPE_SECRET_KEY, STRIPE_PUBLISHABLE_KEY, STRIPE_WEBHOOK_SECRET など）とその設定方法
6. 実行方法（docker-composeを使う場合はそのコマンド、またはdocker直接実行コマンド例）
7. テスト方法（Stripe CLIの使用方法、テストカード番号の例、/ledgerページの確認等）
8. 注意事項（開発用シークレットの扱い、本番では別のシークレットを使うこと、Webhookエンドポイントを本番登録する必要など）
9. **Docker周りの最終調整:** Dockerイメージの最適化やDocker Composeの用意を検討します。本番環境では例えばGunicorn+NGINX構成にする、環境変数はコンテナ外から注入する、SQLiteファイルをボリュームにマウントする、といった調整が考えられます。READMEには開発用手順として簡易なDocker実行例を載せますが、必要に応じてcomposeファイルを用意し、それも説明します。
10. **本番デプロイ準備:** 本番で使用する場合の追加設定を確認します。
11. StripeダッシュボードでWebhookエンドポイント（本番用URL）を登録し、シークレットを取得する。
12. 本番用のStripe公開鍵・秘密鍵を環境変数にセットする（誤ってテストキーを使わないよう区別する）。
13. Webhookの送信イベントとして必要なもの（`checkout.session.completed`など）を選択してあるか確認 [16]。
14. デプロイ先のサーバでポートやHTTPS設定を済ませる。特にWebhookはHTTPSが必須です。ローカルテストではHTTPでしたが、本番では有効な証明書付きのHTTPSエンドポイントである必要があります [17]。
15. スケーリング: 負荷に応じてアプリケーションサーバをスケールアウトする場合、台帳のSQLiteはボトルネックになるため、より強力なデータベース（PostgreSQL等）への移行も検討します。その際はSQLAlchemyなどORMの導入が望ましいです。
16. **最終レビュー:** 同僚やメンターにREADMEの手順で環境を再現してもらい、不明瞭な点や漏れがないかフィードバックをもらいます。READMEに沿ってDay0〜Day5の内容が追体験できれば完了です。

Stripe側の設定や操作:

- **APIキーとWebhookシークレットの区別:** 本番環境用に新たにキーやシークレットを用意する際、テスト用との**切り替え**を誤らないようにします。ダッシュボードで「本番モード」にしてからAPIキーを取得し、環境変数を差し替えます。Webhookシークレットもテスト用と本番用で別物ですので注意してください。
- **料金や税設定:** Checkoutで消費税や送料を扱う場合、Stripe側でTax機能やShipping機能を有効化し、イベントに含まれる`amount_total`と別に`amount_subtotal`や`total_details`などを見る必要があります。今回は単純な例でしたが、ビジネス要件によってはそうした追加情報も台帳に記録すると良いでしょう。
- **カスタマー情報:** Checkout Sessionから顧客メール等を取得できます。本番では顧客ID(Customerオブジェクト)を生成して紐付けることも多いです。Webhookイベントには`customer`フィールド（ID）や`customer_details`（メールや住所）が含まれるので、必要に応じて台帳に保存することも可能です。

確認方法と通過チェックリスト:

- README.mdの手順に沿って、新しい環境でプロジェクトをセットアップし直しても問題なく動作すること（ドキュメントの再現性テスト）。
- 本番運用を想定した設定項目が洗い出され、見落としがないこと。特にStripeのライブモードキーやWebhook設定は誤ると決済が反映されないため、チェックリスト化しておきます。

- 関連者への知識共有: READMEをチームメンバーに配布し、Stripe未経験者でも流れが理解できたというフィードバックを得ます。専門用語には必要に応じ補足説明を加え、コードにもコメントを適宜追加しておきます。
- 以上をもってStripeGymカリキュラムの成果物一式が完成です。あとは必要に応じてリポジトリにタグを打ったり、社内ナレッジベースにリンクを共有したりして、いつでも参照できるようにしましょう。

注意点・トラブルシュート:

- 本番テスト: ライブモードでテストする際は、実際に決済が発生する点に注意してください。少額の商品を用意するか、Stripeのテストモードで徹底的に検証し、本番では一度で済むよう準備します。最初の実取引は社内関係者でテスト購入し、問題なければ一般公開すると安心です。
- ドキュメントの更新: システムに変更を加えたらREADMEも更新する運用にします。特に環境変数や依存サービスが増減した場合は忘れず反映してください。せっかく整備したドキュメントも古くなれば価値が下がるのでメンテナンスも重要です。
- セキュリティ考慮: READMEに機密情報（API秘密鍵やWebhookシークレット）を直接書かないようにします。環境変数の例示もダミー値に留めます。リポジトリ公開時には`.env`を含めない、APIキーをログに出力しないなど、基本的な対策も再確認してください（git-secretsなどのツール活用も検討します）。
- Stripe以外の考慮: 決済完了後の処理はStripeに限定されません。将来的に別の決済手段を追加する場合でも、今回構築したWebhook＋台帳のアーキテクチャは参考になります。汎用的に拡張できるよう、コードやドキュメントを構造化しておくと良いでしょう。

次フェーズに進む前に満たすべき条件:

- 完成したプロジェクトのREADMEおよび補足資料をもとに、Stripe未経験の開発者でも環境構築と基本的なテストができる。
- 本番公開に必要な設定がすべて洗い出され、関係者と共有されている（秘密鍵の管理方法、Webhookの登録方法など）。
- システムは小さな範囲で完成しているので、今後は必要に応じて新機能追加（例: 定期課金Subscription対応やUIの改善）やリファクタリングを進められる状態となった。

以上でStripeGymカリキュラムの全フェーズが完了です。お疲れさまでした！次に、まとめとしてREADMEのテンプレートを掲載します。

README.mdテンプレート

以下は本プロジェクトのREADME.mdのひな形です。適宜プロジェクト名や必要情報を差し替えてご利用ください。

```
**# Stripe Checkout + Webhook + Ledger Demo (Flask + Docker)**

Stripeによる決済(Checkout)とWebhook処理、および簡易台帳管理を組み合わせたデモアプリケーションです。Flaskを使用してサーバサイドを実装し、Docker環境で動作します。**このREADMEでは、環境構築からテスト方法までの手順を説明します。**

**## 機能概要**
- **Stripe Checkoutによる決済ページ:** ユーザはStripeのホストする安全な決済ページでカード決済を行います。
- **Webhook受信による非同期処理:** 決済完了イベントをバックエンドで受信し、自動で支払い情報
```

を記録します。
- **台帳(Ledger)管理:** 支払い情報をSQLiteデータベースに保存し、取引履歴を照会できます（開発用の簡易実装）。

このアプリケーションは学習目的の構成であり、本番環境で使用する際はセキュリティやスケーラビリティ面の追加検討が必要です。

動作環境
- Python 3.x / Flask 2.x
- Docker（Docker Engine が動作する環境）
- Stripe アカウント（テスト用APIキーを使用）
- Stripe CLI（ローカル開発でのWebhook受信テストに使用）

セットアップ手順

1. リポジトリをクローンします。
   ```bash
   git clone https://github.com/your-repo/stripe-flask-ledger.git
   cd stripe-flask-ledger
   ```

2. 環境変数を設定します。プロジェクトルートに`.env`ファイルを作成し、以下のキーを設定してください。
   ```bash
   # .envファイルの例 (実際の値はStripeダッシュボードから取得)
   STRIPE_PUBLISHABLE_KEY=pk_test_XXXXXXXXXXXXXXXXXXXXXXXX
   STRIPE_SECRET_KEY=sk_test_XXXXXXXXXXXXXXXXXXXXXXXX
   STRIPE_WEBHOOK_SECRET=whsec_XXXXXXXXXXXXXXXXXXXXXXXX
   ```
   - **STRIPE_PUBLISHABLE_KEY:** Stripeの公開可能キー（テストモード用）
   - **STRIPE_SECRET_KEY:** Stripeのシークレットキー（テストモード用）
   - **STRIPE_WEBHOOK_SECRET:** StripeのWebhook署名検証用シークレット（Stripe CLI使用時や、ダッシュボードでエンドポイント登録時に入手）

3. Dockerイメージをビルドします。
   ```bash
   docker build -t stripe-flask-ledger:latest .
   ```
   ※初回は時間がかかりますが、キャッシュが有効な限り以降は高速化されます。

4. Dockerコンテナを起動します。
   ```bash
   docker run --env-file .env -p 5000:5000 stripe-flask-ledger:latest
   ```
   上記コマンドでコンテナ内に環境変数が注入され、Flaskサーバがポート5000で起動します。

5. ブラウザでhttp://localhost:5000にアクセスし、ホーム画面が表示されることを確認します。リンク（またはボタン）からStripeのCheckoutページへの遷移を試すことができます。

使い方とテスト

1. テスト用決済の実行
- ホーム画面の「購入」ボタンをクリックすると、Stripeのテスト決済ページにリダイレクトします。
- テストカード番号として「4242 4242 4242 4242」を使用し、適当な有効期限・CVC・郵便番号を入力してください（Stripe提供のテストカード）
- 決済を完了すると、アプリケーションの`/success`ページに戻ります（キャンセルした場合は`/cancel`ページ）。

2. Webhookの受信テスト
StripeからのWebhookをローカルで受け取るには、Stripe CLIを使用します。
1. 新しいターミナルで次を実行し、Stripeイベントを受信転送します:
   ```bash
   stripe listen --forward-to localhost:5000/webhook
   ```
   実行するとWebhook用の署名シークレットキー(`whsec_...`)が表示されます。これを`.env`の`STRIPE_WEBHOOK_SECRET`に追記してください。既に設定済みの場合は表示されたキーと一致するか確認しましょう。
2. 別のターミナルからテストイベントを送信します:
   ```bash
   stripe trigger checkout.session.completed
   ```
   これにより疑似的なCheckout完了イベントが発生し、先ほどのlisten経由でアプリに転送されます。
3. アプリのログ（dockerコンテナの標準出力）にて、イベント受信メッセージが表示されていることを確認してください。例えば `Payment succeeded for Session ...` のようなログです。
4. ブラウザでhttp://localhost:5000/ledgerを開くと、受信したイベントに対応する取引レコードが追記されているはずです。

3. 実際の決済フロー＋Webhook統合テスト
上記1と2の手順を合わせて、実際にブラウザから決済→Webhook処理まで一気通貫でテストします。
- ターミナルで`stripe listen --forward-to localhost:5000/webhook`を実行した状態でブラウザから決済を行います。
- 決済成功後、コンソールログおよび`/ledger`ページで、該当のSessionが記録されたことを確認します。

4. 注意事項
- **Webhookシークレットの管理:** 開発中はStripe CLI発行の一時的なシークレットを使用しています。本番環境ではStripeダッシュボードでWebhookエンドポイントを登録し発行されるシークレットを使用してください。
- **環境変数の管理:** APIキーやシークレットは機密情報です。リポジトリにコミットしないよう注意してください（このプロジェクトでは`.env`を使用し、Git追跡対象外としています）。
- **本番モード:** ライブモードのAPIキーを使うと実際に決済が発生します。テストが十分完了するまで本番キーは絶対に使用しないでください。テストモードでの動作確認が取れてから、環境変数を差し替えて本番用デプロイを行ってください。
- **為替と単位:** 現状JPY固定かつ最小通貨単位（整数）で金額を扱っています。他通貨を扱う場合や小数点が必要な場合、アプリ側ロジックを調整してください。
- **スキーマ変更:** Ledgerテーブルのスキーマは`app.py`内で定義しています。追加で保存したい情報（顧客メール等）があればテーブル定義とINSERT処理を変更してください。
- **スケーリング:** 本アプリはシンプルな実装のため、例えばマルチプロセスで動かすとSQLiteの同時書き込みで競合が起こる可能性があります。高並行環境ではデータベースを他のRDBMSに差し替えることを検討してください。

ファイル構成

├── app.py # Flaskアプリ本体（Stripe Checkoutセッション作成とWebhook処理） ├── requirements.txt # Python依存パッケージ一覧（flask, stripe など） ├── Dockerfile # Dockerビルド用の設定ファイル ├── ledger.db # SQLiteデータベースファイル（初回実行時に生成） └── README.md # 本ドキュメント

※ `ledger.db`は初回起動時に自動生成されます。既存のファイルがある場合、過去のテストデータが残っています。テストの度にクリーンな状態から始めたい場合は、このファイルを削除してください。

ライセンス
本プロジェクトはMITライセンスの下で提供されています。

参考資料
- Stripe公式ドキュメント: [Checkoutの概要](https://stripe.com/docs/payments/checkout) / [Webhookの利用方法](https://stripe.com/docs/webhooks)
- Stripe提供サンプル: [checkout-one-time-payments](https://github.com/stripe-samples/checkout-one-time-payments)
- Flask公式ドキュメント: [Flask Quickstart](https://flask.palletsprojects.com/en/2.2.x/quickstart/)

以上がREADMEテンプレートとなります。実際の値やリンクは適宜あなたのプロジェクトに合わせて修正してください。 [1] [6]

---

[1] [2] [3] [4] Flask Stripe Tutorial | TestDriven.io
https://testdriven.io/blog/flask-stripe-tutorial/

[5] [6] [7] [8] [9] [10] [16] [17] StripeのWebhookについて | Hakky Handbook
https://book.st-hakky.com/hakky/stripe-webhook

[11] google cloud platform - Stripe Error: No signatures found matching the expected signature for payload - Stack Overflow
https://stackoverflow.com/questions/53899365/stripe-error-no-signatures-found-matching-the-expected-signature-for-payload

[12] [14] [15] Fulfill orders | Stripe Documentation
https://docs.stripe.com/checkout/fulfillment

[13] Troubleshooting Stripe Webhook Delivery Issues: A Guide
https://www.woohelpdesk.com/blog/troubleshooting-stripe-webhook-delivery-issues-a-comprehensive-guide/