今回の構成だと **「取得 ⇒ 変換 ⇒ 出力」を“配管”する役**を Camel が担当しています。

このプロジェクトでの Camel の役割

- **ルーティング（EIP 実装）**
  - `direct:run` を入口にして、 **HTTP 取得 → Processor で PDF→CSV 変換 → ファイル出力** の順に“配線”。
  - ルートは 1 本のパイプラインとして記述され、I/O の面倒は Camel が見るので、アプリ側は変換ロジック（`PdfToCsvProcessor`）に集中できます。
- **コンポーネント利用**
  - `camel-http`：`app.sourceUrl`（compose では `http://pdf-host:10081/...`）から PDF を取得。
  - `camel-file`：`/data/output/…csv` に保存（エラーファイル先も指定可）。
  - `bean/processor`：`PdfToCsvProcessor` を挟み、**ボディを CSV 文字列に変換**。
- **動的エンドポイントと環境差分吸収**
  - `.toD("{{app.sourceUrl}}?bridgeEndpoint=true")` のように **プロパティ置換**。 Docker/ローカルで URL や出力先を **設定だけ**で切替できます（コード修正不要）。
- **例外ハンドリング／リトライ（onException で拡張可能）**
  - ルートの先頭に `onException()` を置けば、**再試行・バックオフ・デッドレター**などを集中定義可能。 いまは簡素ですが、運用段階で**一時的な HTTP 失敗の自動再試行**や**失敗時の別ディレクトリ退避**を足すのが定石です。
- **同期呼び出しの隠蔽（ProducerTemplate）**
  - Controller から `ProducerTemplate.requestBody("direct:run", …)` で**ルートを関数のように呼べる**。 Web レイヤは Camel の背後の I/O を意識せず、完了／失敗だけ見ればOK。
- **監視・ログの一元化**
  - ルートに `.log()` を差し込むだけで**途中の状態**を観測可能。必要に応じて**メトリクス（Micrometer）**にも出せます。

流れ（ざっくりシーケンス）

```
POST /run
  └─ ProducerTemplate → direct:run (Camel)
       ├─ HTTP( camel-http ) GET app.sourceUrl → PDF bytes
       ├─ Processor PdfToCsvProcessor → CSV string
       └─ file: /data/output/jsprice_20250630.csv ( 必要なら /data/error へ)
```

Camel を使うメリット（この用途での実感値）

- **I/O の再配線が速い**：HTTP を S3・FTP・ローカルファイルに差し替えるのもルート1行変更で済む。
- **運用要件に強い**：リトライ／DLQ／並列化／スケジューリングなどを**アプリ本体を汚さず**足せる。
- **テスト容易**：`AdviceWith` や `CamelTestSupport` で外部 I/F をスタブ化してルート単体テストが書ける。

ここから足せる“Camel らしい”拡張アイデア

- **スケジューラ実行**：`from("timer:jsprice?period=1h")` → 定期取得。
- **フォールバック**：HTTP 失敗時に別 URL（ミラー）へ切替（**Content Based Routing**）。
- **複数ファイル処理**：`from("file:/inbox?include=*.pdf&move=.done")` → 連続変換。
- **並列化**：`.split(body()).parallelProcessing()` 等で大量 PDF をスループット向上。
- **監査ログ**：`.wireTap("file:/data/audit")` で非同期に処理記録を保存。

> まとめると、Camel は **「接続（HTTP/FILE）・制御（再試行/並列）・拡張（差し替え）」を統一的に面倒を見る**ミドルウェア。 変換ロジックは Processor に閉じ込め、周辺の“運用の強さ”は Camel で後追い強化できる —— これが今回のプロジェクトでの活躍ポイントです。