

Day3：退勤機能と業務ルールの実装

状態遷移と排他制御 (Service層の責務)

本日のゴール

- 「退勤」ボタンを実装し、業務フローを完成させる
- 状態遷移（未出勤 → 出勤中 → 退勤済み）を理解する
- 不正な操作（出勤前の退勤など）をエラーにする



業務ルールと状態遷移

未出勤 (Not Worked)

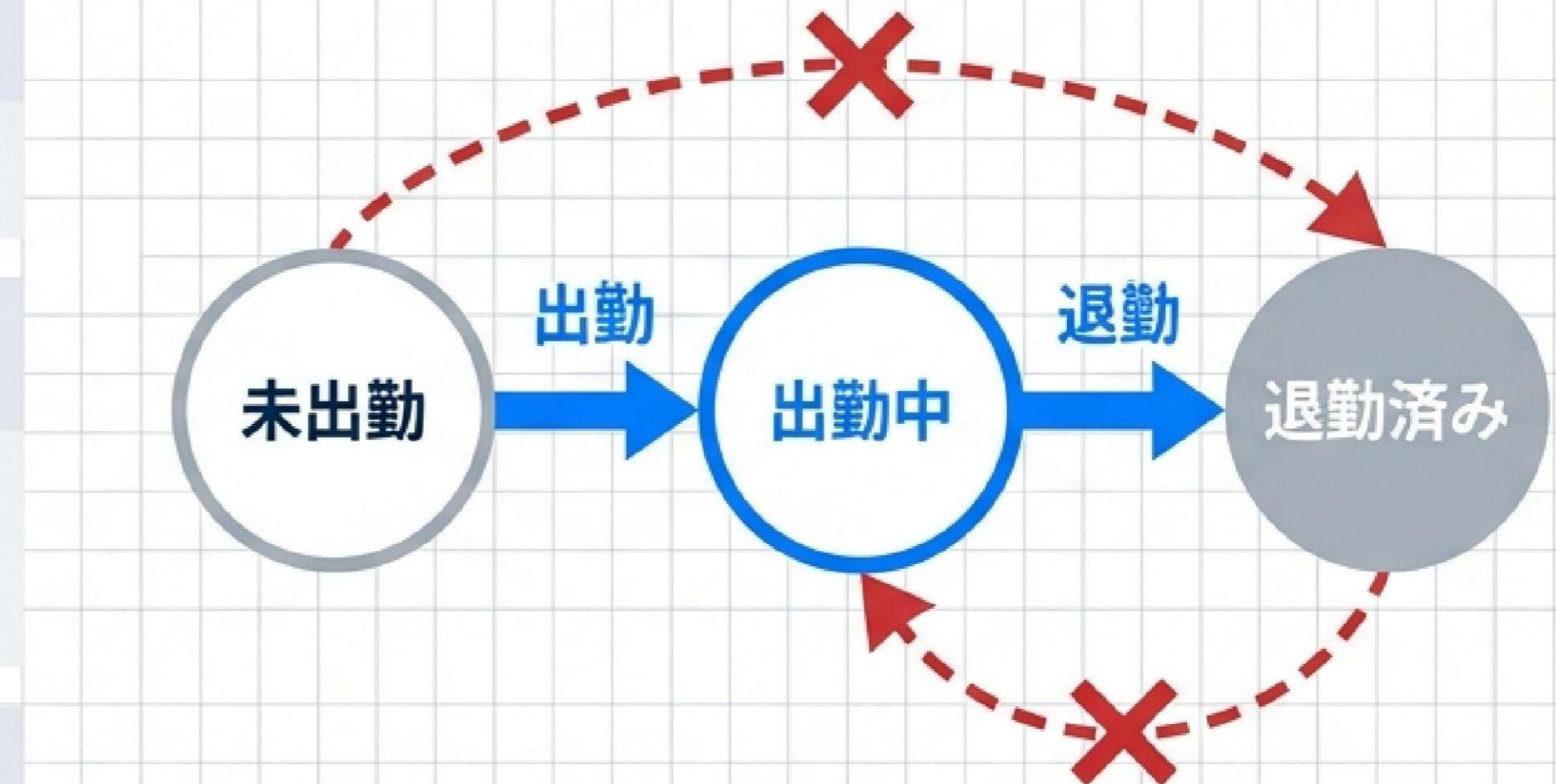
「出勤」のみ可能。「退勤」は不可。

出勤中 (Working)

「退勤」が可能。「出勤」は不可（二重出勤NG）。

退勤済み (Finished)

本日の業務終了。操作不可。



実装の流れ

1

Setup: Day2のコードを複製
Noto Sans JP Regular

2

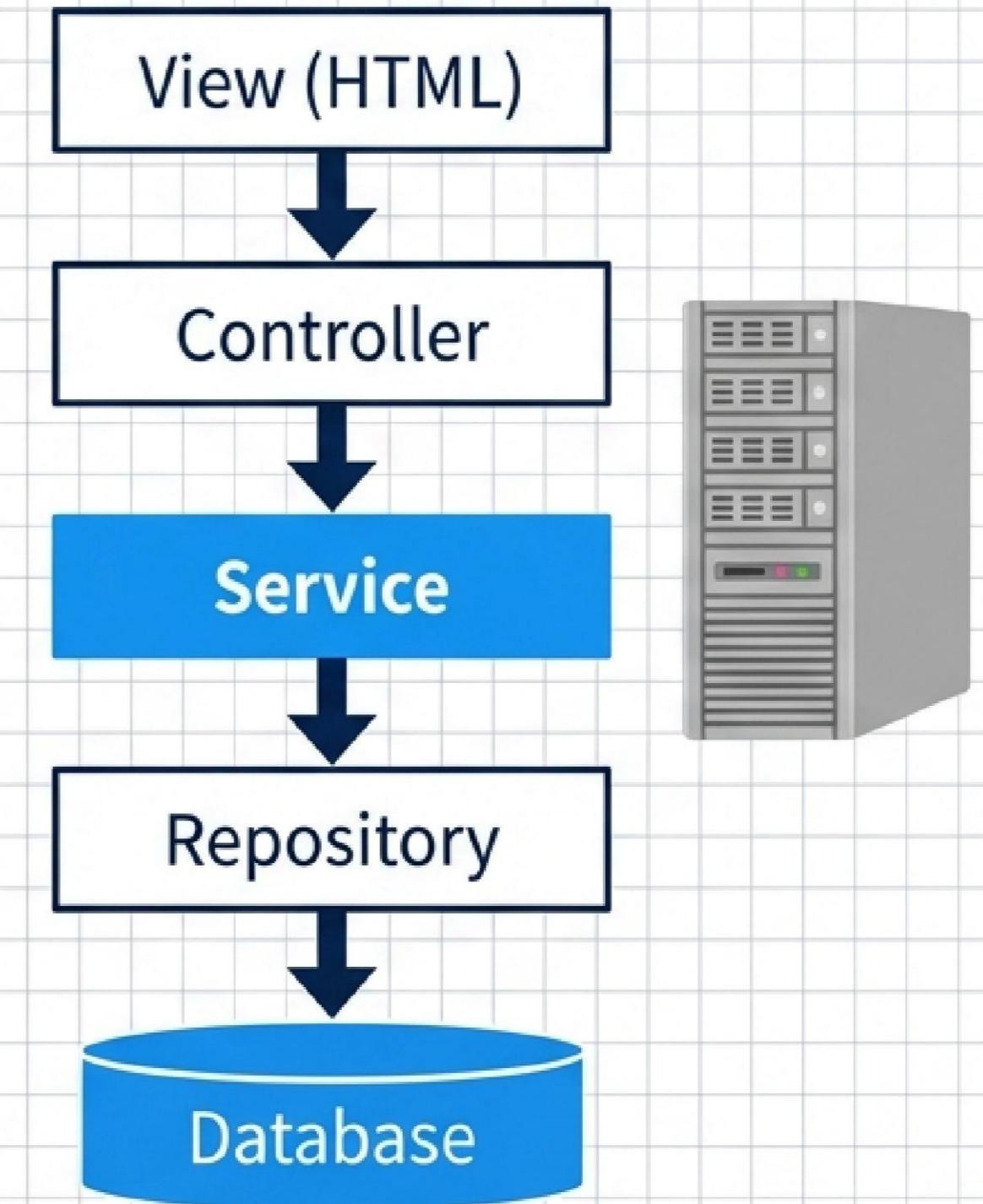
Service: 業務ロジックの実装（ルール定義）
Noto Sans JP Regular

3

Controller: リクエストの受付と応答
Noto Sans JP Regular

4

View: ボタンの表示制御
Noto Sans JP Regular



Service - 退勤ロジック

チェック処理 (Guard Clauses):

1. 当日のデータが存在するか?
(なければエラー)
2. すでに退勤済みステータスか?
(ならばエラー)

正常系の処理:

- ステータスを「退勤済み」に更新
- 現在時刻をセット

```
① AttendanceService.java ×

public void clockOut(int userId) {
    // 1. データ存在チェック
    Attendance attendance = repository.findByIdAndDate(userId, LocalDate.now())
        .orElseThrow(() -> new BusinessException("出勤していません"));

    // 2. ステータスチェック (ガード節)
    if (attendance.getStatus() == AttendanceStatus.FINISHED) {
        throw new BusinessException("すでに退勤済みです");
    }

    // 更新処理
    attendance.setEndTime(LocalTime.now());
    attendance.setStatus(AttendanceStatus.FINISHED);
    repository.save(attendance);
}
```

Controller - リクエスト処理

- エンドポイント:
`@PostMapping("/clock-out")`
- Service呼び出し:
`service.clockOut()`を実行
- Viewへのデータ渡し:
ボタンの表示制御フラグ
`canClockOut`をModelに
セット

HomeController.java

```
@PostMapping("/clock-out")
public String clockOut(Model model) {
    try {
        service.clockOut(1); // ユーザーID固定
    } catch (BusinessException e) {
        model.addAttribute("errorMessage", e.getMessage());
    }
    return "index";
}
```

View - ボタンの出し分け

Thymeleafの条件分岐 (`th:if`)

- サーバー側で判定した
フラグを使用する
- `canClockIn` が true
→ 出勤ボタンを表示
- `canClockOut` が true
→ 退勤ボタンを表示

index.html

```
<!-- 出勤ボタン -->
<form th:if="${canClockIn}" action="/clock-in" method="post">
    <button type="submit">出勤</button>
</form>

<!-- 退勤ボタン -->
<form th:if="${canClockOut}" action="/clock-out" method="post">
    <button type="submit" class="btn-clock-out">退勤</button>
</form>
```

CSS - スタイルの調整

UX (ユーザーエクスペリエンス) の向上

- 重要なアクション（出勤・退勤）は色で区別する
- 誤操作を防ぐための視覚的フィードバック
- 退勤ボタンには警告色（赤/ピンク系）を採用

styles.css

```
/* 退勤ボタン用クラス */
.btn-clock-out {
    background-color: #ef4444; /* Red-500 */
    color: white;
}

.btn-clock-out:hover {
    background-color: #dc2626; /* Darker Red */
}
```

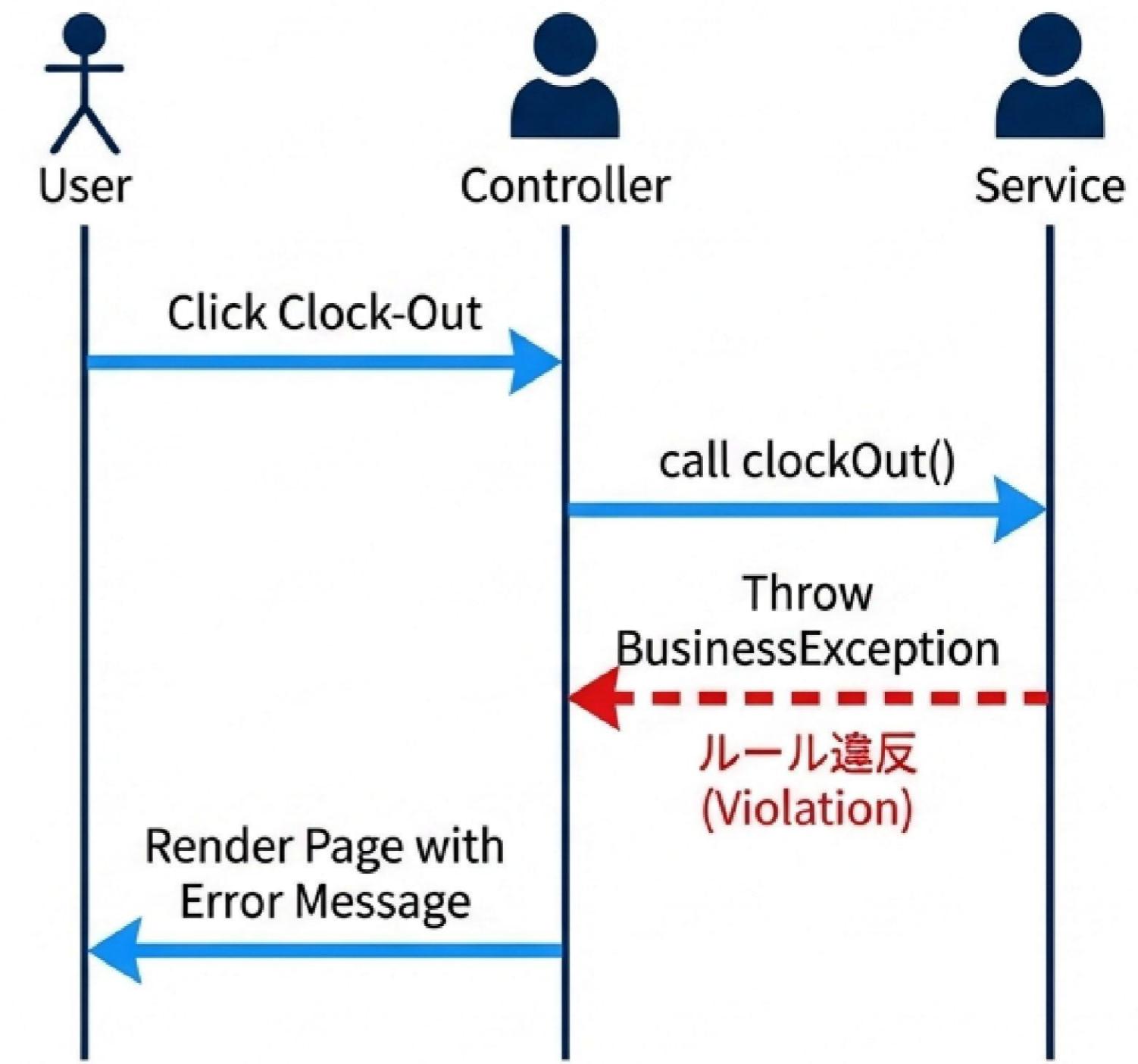
出勤

退勤

エラーハンドリングの仕組み

例外 (Exception) のバケツリレー

- 1. Service:** ルール違反を検知 → `BusinessException` を投げる (Throw)
- 2. Controller:** 例外を捕捉 → `catch` してエラーメッセージを取り出す
- 3. View:** `th:if` でメッセージがあればアラートを表示



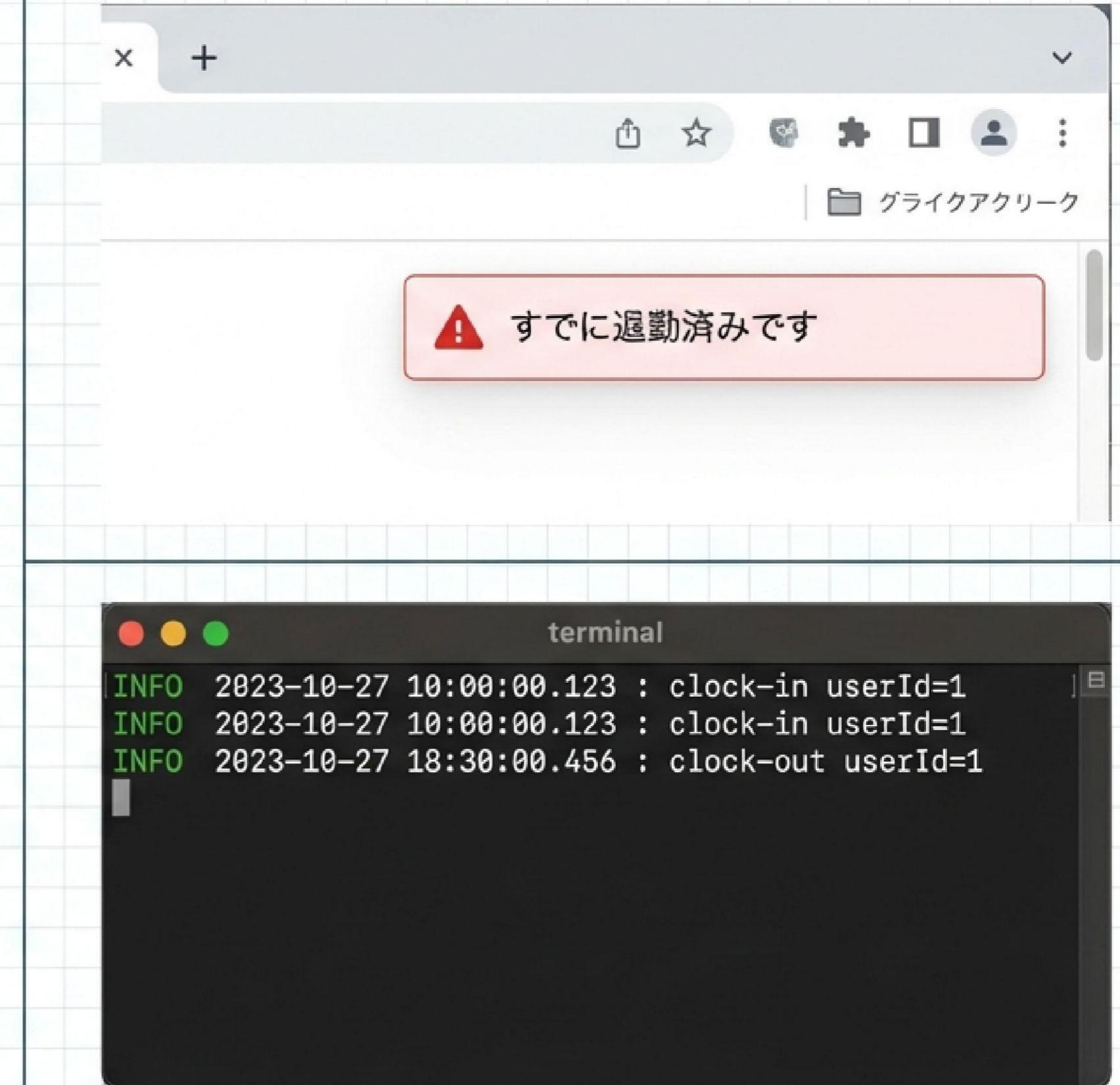
動作確認とログ

検証シナリオ:

1. 正常系: 出勤 → 退勤 (ステータスが「退勤済み」になること)
2. 異常系: 出勤せずに退勤、または退勤後に再度退勤 (エラーが出ること)

ログ確認:

- ターミナルで `INFO` ログが出力されているか確認する。



すでに退勤済みです

```
terminal
INFO 2023-10-27 10:00:00.123 : clock-in userId=1
INFO 2023-10-27 10:00:00.123 : clock-in userId=1
INFO 2023-10-27 18:30:00.456 : clock-out userId=1
```

まとめ：Service層の重要性

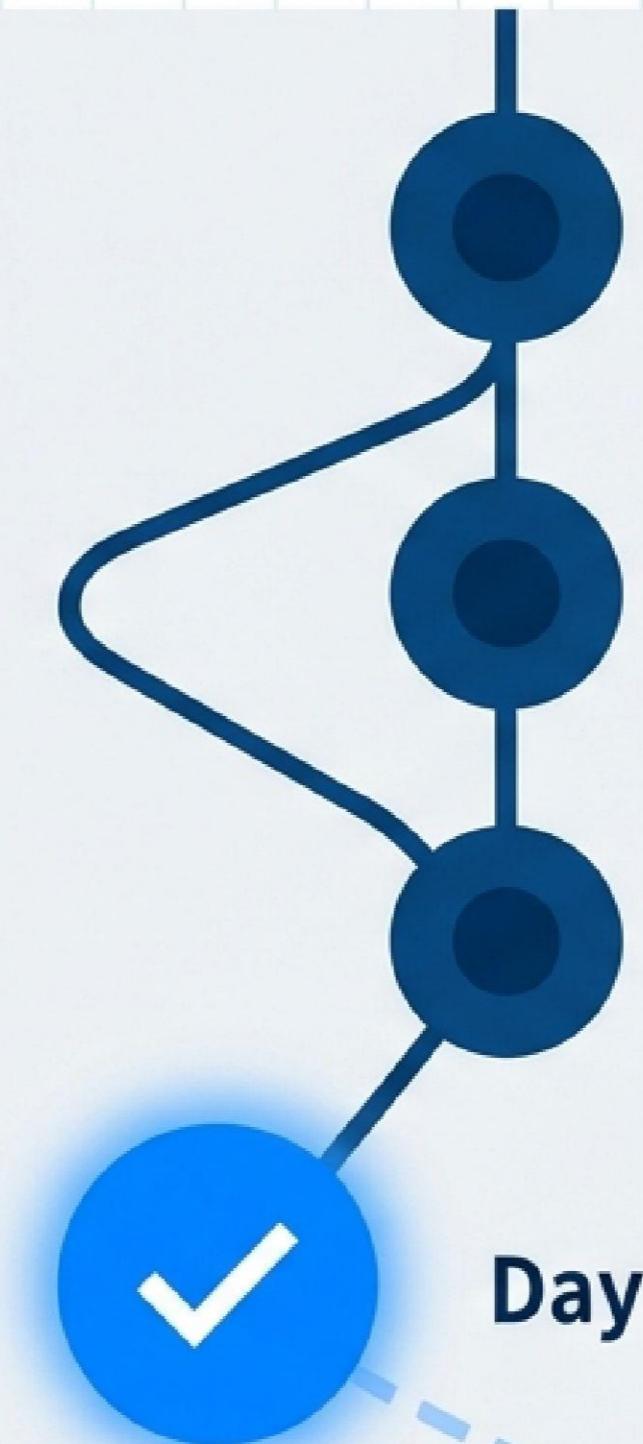
- 業務ルール（ビジネスロジック）は **Service** に集約する。
- Controllerは「交通整理」、Viewは「表示」に専念する。
- 役割分担（MVC）を守ることで、変更に強いコードになる。



研修全体の振り返り

- Day 0: 事前学習 (HTML/Java基礎)
- Day 1: 環境構築とMVC基礎 (表示のみ)
- Day 2: データベース連携 (出勤機能)

✓ Day 3: 業務ルールの実装 (退勤・排他制御) [DONE]



Day 0: 事前学習

Day 1: 環境構築

Day 2: データベース

Day 3: 業務ルール

Next: Docker/Deploy