

Day2：出勤機能の実装とDB連携

Entity, Repository, Service の役割

新入エンジニア向けプログラミング研修

本日のゴール

- ✓ 「出勤」ボタンを押して、データベース(DB)にデータを保存できる。
- ✓ Controller → Service → Repository → DB の連携フローを理解する。
- ✓ 業務ルール（1日1回しか出勤できない）を実装する。



アーキテクチャの全体像

Controller (受付)

リクエストを受け取り、Serviceへ指示を出す。

Service (業務ロジック)

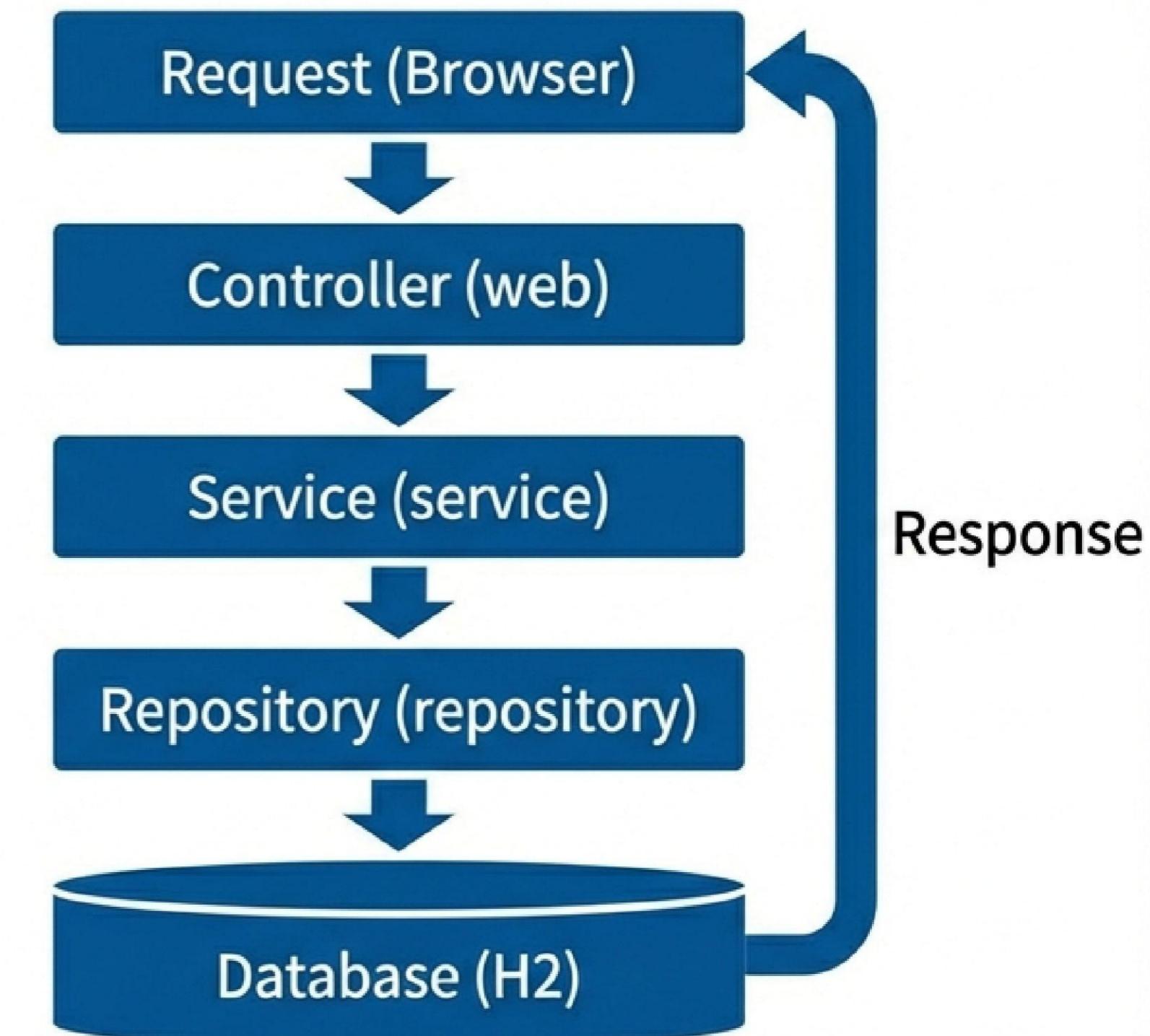
ルール（例：二重出勤チェック）を確認・判断する。

Repository (DB操作)

データベースへの保存・検索を行う。

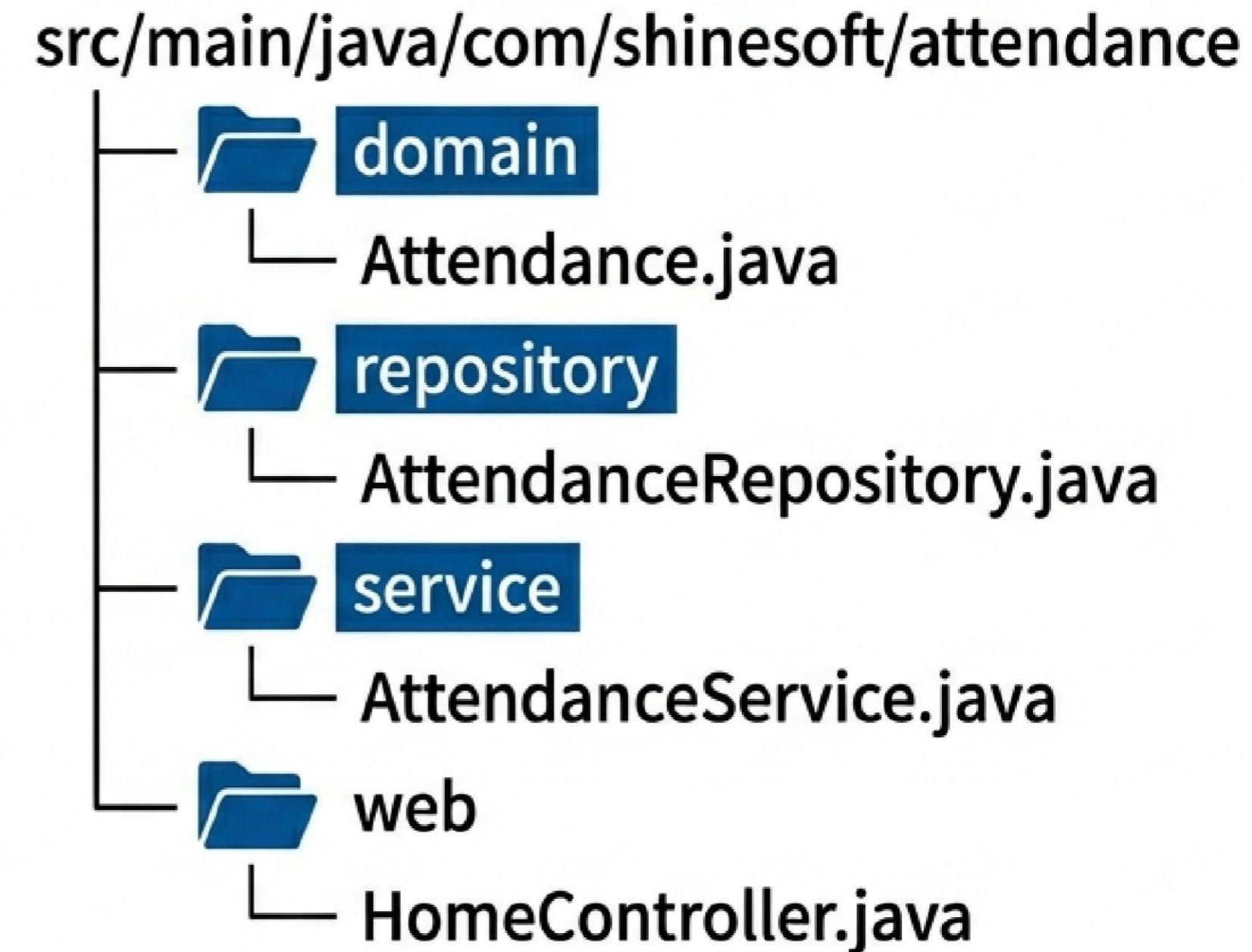
Database

実際のデータを保管する場所。



作成するファイルとパッケージ

- domain: データの「型」を定義 (Entity)
- repository: DBアクセスのためのインターフェース
- service: 具体的な業務処理の実装
- web: 画面とやり取りするコントローラー

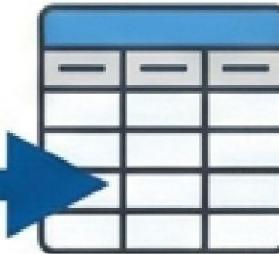


Domain (Entity) - データの型

- `@Entity`: このクラスがDBのテーブルであることを示す。
- `@Id`: 主キー (Primary Key) となる項目。
- `@GeneratedValue`: IDを自動採番 (オートインクリメント) する設定。
- `@Table`: テーブル名や制約 (ユニーク制約など) を指定。

```
@Entity
@Table(name = "attendance")
public class Attendance {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Integer id;

    private LocalDateTime startTime;
    // ...
}
```



Repository - DB操作

JpaRepository

- Springが提供する魔法のインターフェース。
- メリット: SQLを書かずに `save()` (保存) や `findAll()` (全検索) などのメソッドが使えるようになる。
- 継承: `extends JpaRepository<Entityクラス, IDの型>` と書くだけで機能する。

```
@Repository
public interface AttendanceRepository
    extends JpaRepository<Attendance,
    Integer, Integer> {

    // 宣言だけでDB操作が可能になる
}
```

Service - 業務ロジック

- `@Service`: 業務ロジックを行うクラス。
- `@Transactional`: 処理の一貫性を保証。

二重出勤チェックのロジック:

1. 今日の日付で既にデータがあるか確認
2. あればエラー (BusinessException) を投げる
3. なければ `repository.save()` で保存

```
@Service
@Transactional
public class AttendanceService {
    public void clockIn(User user) {
        if (repository.existsByUserAndDate(user, LocalDate.now())) {
            throw new BusinessException("出勤済みです");
        }
        repository.save(new Attendance(user));
    }
}
```

初期データ投入 (DataSeeder)

- ・目的: アプリ起動時に、テスト用のユーザーデータを自動生成する。
- ・CommandLineRunner: アプリ起動完了直後に実行されるインターフェース。
- ・User: 今回は固定ユーザー (user1) を作成してDBに保存しておく。

```
@Component
public class DataSeeder implements CommandLineRunner {
    @Override
    public void run(String... args) {
        if (userRepository.count() == 0) {
            User user = new User("user1");
            userRepository.save(user);
        }
    }
}
```

Controller - 画面とロジックの橋渡し

- Dependency Injection (DI):
final フィールドと
@RequiredArgsConstructor で
Service を利用可能にする。
- @PostMapping: フォームからの
送信を受け取る。

処理フロー:

1. service.clockIn() を呼び出す。
2. 成功したらトップページ
(redirect:/) へ戻る。

```
@PostMapping("/clock-in")
public String
public String clockIn() {
    User user = //... user1を取得
    attendanceService.clockIn(user);
    return "redirect:/";
}
```

View (Thymeleaf) - ボタンと表示

- **th:if / th:unless:** 条件によって表示内容を切り替える。
 - 未出勤なら「ボタン」
 - 出勤済みなら「時刻」
- **th:text:** サーバーから渡されたデータを表示。
- **Form:** **<form>** でControllerへリクエストを送る。

```
<!-- 未出勤の場合 -->
<div th:if="${status == 'unworked'}">
  <form th:action="@{/clock-in}" method="post">
    <button>出勤</button>
  </form>
</div>

<!-- 出勤済みの場合 -->
<div th:if="${status == 'working'}">
  <p th:text="${startTime}"></p>
</div>
```

CSS - 見た目の調整

- UIフィードバック: 成功時やエラー時の見た目を整える。
- .alert-danger: エラーメッセージを赤枠で目立たせる。
- ボタン: クリックしやすいサイズと色（青）を定義。

```
.alert-danger {  
  color: #721c24;  
  background-color: #f8d7da;  
  border-color: #f5c6cb;  
}  
  
button {  
  background-color: var(--accent);  
  color: white;  
}
```

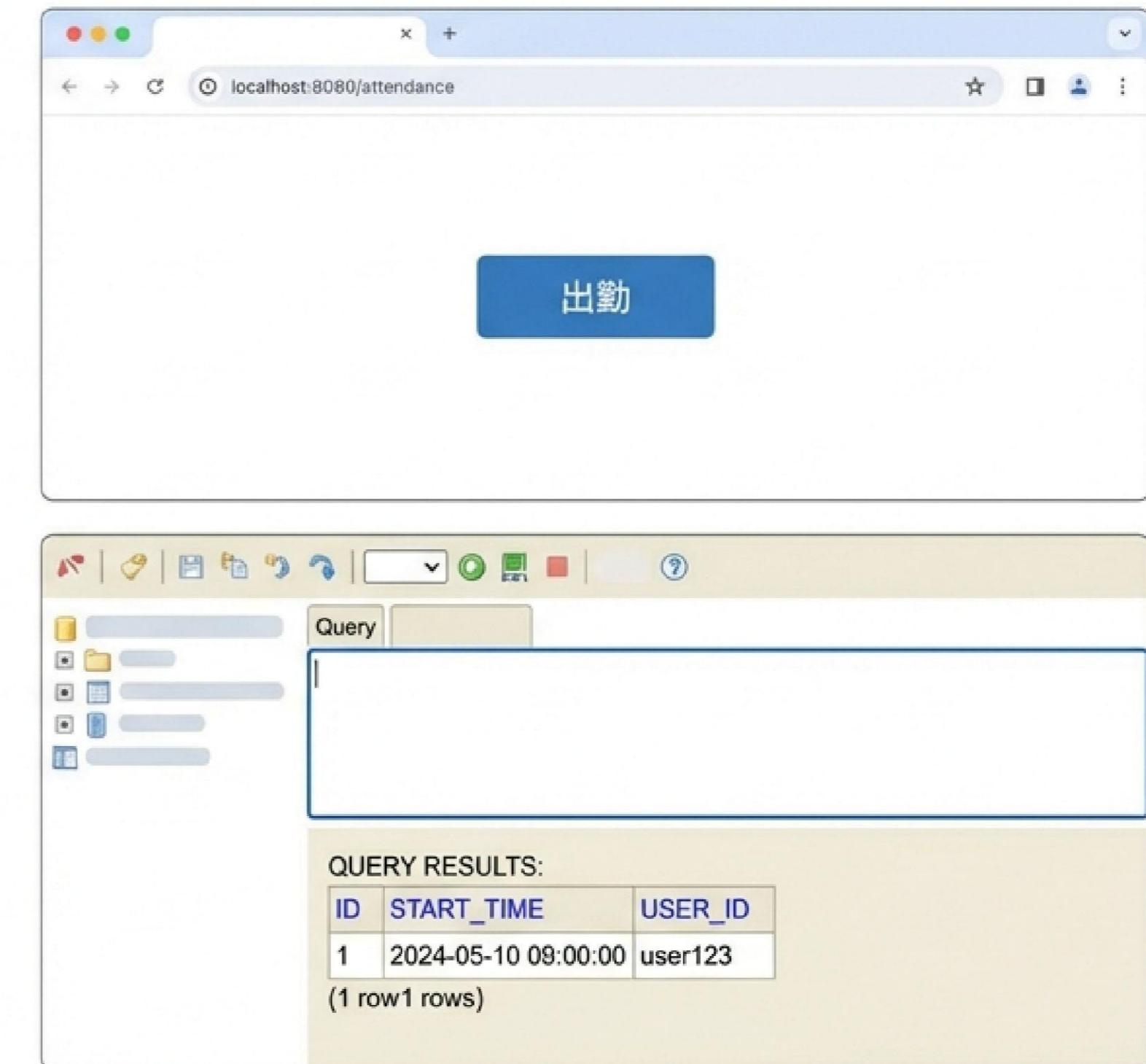
Error

Button

動作確認 (Verification)

H2 Console (DB確認)

- URL: /h2-console にアクセス。
- 確認手順: SELECT * FROM ATTENDANCE; を実行。
- 期待値: ボタンを押した後、レコードが1行増えていること。



The screenshot shows the H2 Console interface. At the top, a browser window displays a simple web page with a blue button labeled "出勤" (Attendance). Below this, the H2 Console interface is shown, featuring a toolbar, a file tree on the left, and a query editor on the right. The query editor contains the SQL command: "SELECT * FROM ATTENDANCE;". The results pane shows a table with one row of data:

ID	START_TIME	USER_ID
1	2024-05-10 09:00:00	user123

Below the table, the text "(1 row 1 rows)" is displayed, indicating that one row was inserted into the database.

エラーハンドリング

操作と動作

- **操作:**
- すでに出勤している状態で、もう一度「出勤」ボタンを押す。
- **動作:**
 1. ServiceがBusinessExceptionを投げる。
 2. Controllerがそれをキャッチ(try-catch)。
 3. 画面にエラーメッセージを表示。



まとめ (Summary)

- **MVC + Service/Repository**: Webアプリの基本構造を実装しました。
- **DB連携**: EntityとRepositoryを使って、データを永続化（保存）しました。
- **業務ロジック**: 「1日1回」というルールをService層で実装しました。

Next Step (Day 3):

次は「退勤」機能と、過去の履歴を表示する「一覧」機能を作成します。