勤怠管理システム 設計書

---

1. システム概要

システムの目的と概要

この勤怠管理システムは、シンプルなREST APIを通じてユーザー管理と勤怠（出勤・退勤）打刻、月次レポート出力を行うための軽量サービスです。
学習／演習用途を想定した最小構成で実装されており、以下の要素を備えます。

- ユーザーの作成・一覧取得
- 出勤（チェックイン）・退勤（チェックアウト）の打刻
- 月次勤怠レポートの取得
- 健康チェック用ヘルスエンドポイント
- SQLite + SQLAlchemy による永続化
- CLI による DB 初期化とサンプルデータ投入

主な機能一覧

- ユーザー管理
    - `POST /api/users`：ユーザー作成
    - `GET /api/users`：ユーザー一覧取得
- 勤怠管理
    - `POST /api/attendances/checkin`：出勤打刻
    - `POST /api/attendances/checkout`：退勤打刻
- レポート
    - `GET /api/reports/monthly`：特定ユーザーの月次勤怠情報取得（集計含む）

- 運用ユーティリティ
    - `GET /health`：ヘルスチェック
    - CLIフラグ `--init-db`：DB初期化＋サンプルデータ投入

---

2. アーキテクチャ設計

使用技術（フレームワーク・DB・構成図）

- アプリケーションフレームワーク：Flask（WSGI）
- ORM：SQLAlchemy
- データベース：SQLite（ファイル：`attendance.db`）
- マイグレーション（推奨）：Flask-Migrate（requirements に含まれる想定）
- 入力バリデーション：軽量ユーティリティ＋例外処理（marshmallow 等を導入可能）
- 実行方式：開発モードは `app.run()`。本番では WSGI サーバ（gunicorn など）を推奨。

簡易構成図（テキスト）

```
[Client (ブラウザ/APIクライアント)]
        |
     HTTP(S)
        |
[Flask アプリケーション] -- ロジック / ルーティング / バリデーション
        |
  SQLAlchemy ORM
        |
   [SQLite ファイル DB]
```

コンポーネント構成（概念）

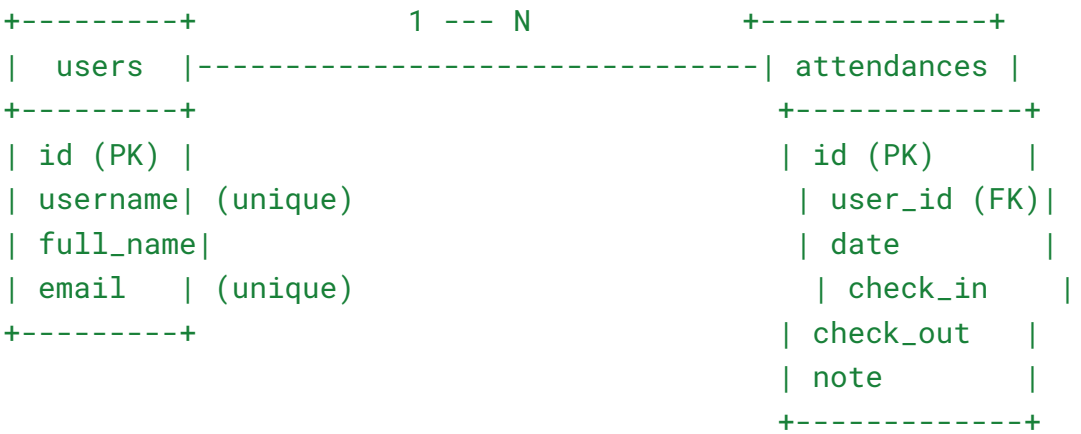
- プレゼンテーション層（HTTP API）
    - ルート定義（/api/users, /api/attendances/*, /api/reports/*, /health）

    - エラーハンドリング（abort を使った HTTP ステータス返却）
- アプリケーション層（ユースケース）
    - ユーザー作成・一覧
    - 打刻ロジック（チェックイン／チェックアウトの重複防止や存在チェック）
    - レポート生成（フィルタ・集計）
- ドメイン／永続化層
    - モデル：User、Attendance（SQLAlchemy モデル）
    - DB 制約：ユニーク制約（ユーザー名・メール、user_id + date のユニークなど）
- オペレーション
    - CLI 初期化（DB 作成・サンプル投入）
    - ヘルスチェックエンドポイント

---

3. データベース設計

ER図（テキスト表現）



テーブル定義（主要カラム）

users

- `id` : INTEGER, PRIMARY KEY, NOT NULL
- `username` : VARCHAR(80), UNIQUE, NOT NULL
- `full_name` : VARCHAR(120), NOT NULL
- `email` : VARCHAR(120), UNIQUE, NOT NULL

制約・備考：`username` と `email` にユニーク制約。ユーザー識別は `username` を想定。

attendances

- `id` : INTEGER, PRIMARY KEY, NOT NULL
- `user_id` : INTEGER, FOREIGN KEY → users.id, NOT NULL
- `date` : DATE, NOT NULL (YYYY-MM-DD)
- `check_in` : TIME, NULLABLE (HH:MM)
- `check_out` : TIME, NULLABLE (HH:MM)
- `note` : VARCHAR(255), NULLABLE

制約・備考：

- `UNIQUE(user_id, date)`：同一ユーザー・同一日に複数レコードが作られないようにする。
- 外部キー制約は ORM 側で管理（DB 側に FK 制約を張るかは設定次第）。

---

4. 機能設計

> 以下は各 API の入出力仕様（シンプルな JSON ベース）。ステータスは標準的な HTTP ステータスを返却する設計。

共通事項

- リクエスト・レスポンスは `application/json`

- 日付フォーマット：`YYYY-MM-DD`
- 時刻フォーマット：`HH:MM`（24時間）
- エラーは HTTP ステータス + エラーメッセージ（文字列）で返却

4.1 `POST /api/users` — ユーザー作成

- 説明：新しいユーザーを作成する
- リクエストボディ（JSON）

```
{
  "username": "taro",
  "full_name": "山田 太郎",
  "email": "taro@example.com"
}
```

- 成功レスポンス（201 または 200）

```
{
  "user": {
    "id": 1,
    "username": "taro",
    "full_name": "山田 太郎",
    "email": "taro@example.com"
  }
}
```

- エラーパターン
  - 400 Bad Request：必須パラメータ不足（username/full_name/email）
  - 400 Bad Request：username または email が重複
  - 500 Internal Server Error：DB エラー等

4.2 `GET /api/users` — ユーザー一覧取得

- 説明：登録済みユーザーを全件取得
- パラメータ：なし（将来的にページネーションや検索クエリを追加可能）
- 成功レスポンス（200）

```
{
  "users": [
    { "id":1, "username":"taro", "full_name":"山田 太郎",
"email":"taro@example.com" },
    ...
  ]
}
```

4.3 `POST /api/attendances/checkin` — 出勤打刻

- 説明：指定ユーザーの指定日に出勤時刻（check_in）を登録
- リクエストボディ（JSON）

```
{
  "username": "taro",
  "date": "2025-10-01",
  "time": "09:00"
}
```

- 処理の流れ（ロジック）
    1. `username`, `date`, `time` の存在チェック（必須）
    2. `username` からユーザー取得 → 存在しない場合 404
    3. `date`/`time` の形式検証（`%Y-%m-%d` / `%H:%M`）
    4. `attendances` テーブルで `(user_id, date)` を検索
        - レコードがなければ新規作成（check_in に時刻をセット）

        - 既に存在していて `check_in` がセット済みなら 400（重複打刻防止）
        - 存在しているが `check_in` が空なら更新してセット
    5. commit → 成功レスポンス：登録済み勤怠オブジェクト（200）
- 成功レスポンス（200）

```
{ "attendance": { "id": 10, "user_id":1, "date":"2025-10-01",
"check_in":"09:00", "check_out": null, "note": null } }
```

- エラーパターン
    - 400：必須パラメータ不足
    - 400：日付/時刻フォーマット不正
    - 404：ユーザー未登録
    - 400：既にチェックイン済み

4.4 `POST /api/attendances/checkout` — 退勤打刻

- 説明：指定ユーザーの指定日に退勤時刻（check_out）を登録
- リクエストボディ（JSON）

```
{
  "username": "taro",
  "date": "2025-10-01",
  "time": "18:00"
}
```

- 処理の流れ（ロジック）
    - 入力チェック（username, date, time）
    - ユーザー存在確認

    - 日付・時刻の形式検証
    - `(user_id, date)` の勤怠レコード取得
        - レコードがない場合は 404 または（要設計）新規作成後に check_out をセットする仕様も検討可
        - 既に `check_out` がセット済みなら 400（重複防止）
        - 正常なら `check_out` をセットして commit
    - 成功レスポンス：更新した勤怠オブジェクト（200）
- エラーパターン
    - 400：必須パラメータ不足／フォーマット不正／既にチェックアウト済み
    - 404：ユーザー未登録／勤怠レコードなし（仕様次第で 404 または記録自動作成）

4.5 `GET /api/reports/monthly` — 月次レポート

- 説明：指定ユーザー・指定月の勤怠一覧・集計を取得
- クエリパラメータ（例）
    1. `username`（必須）
    2. `year`（例：2025、必須）
    3. `month`（例：10、必須）
- 処理の流れ（ロジック）
    1. パラメータ検証
    2. ユーザー存在確認
    3. 指定月の `attendances` を取得（`date` BETWEEN yyyy-mm-01 AND yyyy-mm-last）
    4. 各日ごとの出退勤を列挙し、月次の集計（出勤日数、総実働時間（見積り）、欠勤日等）を計算して返却

- 成功レスポンス（200）例

```
{
  "username":"taro",
  "year":2025,
  "month":10,
  "attendances":[

{"date":"2025-10-01","check_in":"09:00","check_out":"18:00","note":n
ull},
    ...
  ],
  "summary":{
    "work_days":20,
    "total_work_hours":"160:30"  // 実装次第で秒単位・分単位の整数も可
  }
}
```

- エラーパターン
    - 400：パラメータ不足・不正
    - 404：ユーザー未登録

4.6 `GET /health` — ヘルスチェック

- シンプルな生存確認。`{ "status": "ok" }` を返却。

バリデーションとエラー処理方針

- 入力バリデーション
    - すべての API は必須パラメータを明示的にチェックすること。
    - 日付（`YYYY-MM-DD`）・時刻（`HH:MM`）はパーサで厳密に検証し、不正フォーマットは 400 を返却。

    - 文字列長等の基本的な制約は ORM（カラム定義）側でも担保。

- エラー分類
    - 400 Bad Request：入力不備、フォーマット不正、業務ルール違反（重複打刻など）
    - 404 Not Found：ユーザーや勤怠レコードが存在しない場合
    - 409 Conflict（将来的に採用推奨）：重複リソース作成時の競合
    - 500 Internal Server Error：予期しない例外、DB 接続障害など（ログ出力）
- レスポンス設計
    - エラー発生時は短いメッセージを返す（日本語メッセージが実装されているためログと一致させる）
    - 本番では詳細な例外メッセージは抑制し、ログに詳細出力すること

---

5. 業務フロー

ユースケース: ユーザー登録 → 勤怠記録 → 月次レポート出力

```
[ユーザー（従業員）]
    |
    |--(1) ユーザー登録リクエスト→ POST /api/users
    |         (username, full_name, email)
    |
[システム] ユーザー作成（users テーブルに row 作成）
    |
    |--(2) 出勤 : POST /api/attendances/checkin
    |         (username, date, time)
    |         - ユーザー検索（username）
    |         - (user_id, date) のレコードを取得／作成
    |         - check_in に時刻を保存
    |
    |--(3) 退勤 : POST /api/attendances/checkout
    |         (username, date, time)
    |         - (user_id, date) のレコード取得
    |         - check_out に時刻を保存
    |
```

```
    |--(4) レポート : GET
/api/reports/monthly?username=...&year=...&month=...
    |          - 指定月の attendances を取得
    |          - 日別データ + 集計（出勤日数、合計実働時間等）を算出して返却
```

視覚的な手順（簡易フローチャート・テキスト）

1. 管理者またはユーザーが `POST /api/users` で登録
2. ユーザーは出勤時に `checkin`、退勤時に `checkout` を呼ぶ
3. DB は `attendances` に日別で 1 レコードを保持（`UNIQUE(user_id,date)`）
4. 月次レポートはその日別データを集計して提供

---

6. 非機能要件

セキュリティ

- 認証・認可：現在サンプル実装は認証無し。実運用では以下を必須導入
    - トークンベース認証（JWT）または OAuth2（Resource Owner Password などは非推奨）
    - API キー／Bearer トークンでエンドポイント保護
    - 管理系 API はロール（管理者）によるアクセス制御
- 入力検証：SQL インジェクションは ORM により緩和されるが、パラメータ検証は厳格に行う
- 通信の保護：TLS（HTTPS）を必須化
- 情報漏えい対策：エラーメッセージに内部情報を出さない／ログに機密情報を出さない

パフォーマンス

- 現状 SQLite（単一ファイル）は軽量で開発用に適するが、並列アクセスが増えるとボトルネックになるため本番は RDBMS（PostgreSQL 等）を推奨

- 高負荷想定時は DB コネクションプール、キャッシュ層（Redis）導入を検討
- レポートは日次・月次集計を行うため、集計クエリの最適化（インデックス：`attendances(user_id, date)`）を推奨

可用性・運用・保守

- マイグレーション管理：Flask-Migrate（Alembic）でスキーマ変更を管理
- バックアップ：定期的な DB バックアップ（本番では RDBMS のスナップショット）
- 監視：ヘルスチェック（/health）、ログ監視、メトリクス（Prometheus など）を導入
- ロギング：操作ログ（誰がいつ打刻したか）および エラーログの仕組みを整備
- テスト：単体テスト・統合テスト（Flask のテストクライアント）を整備
- 運用コマンド：CLI による初期化・サンプル生成（`--init-db`）があるため自動化スクリプトとして CI に組み込める

拡張性（設計上の注意）

- 将来的に「勤怠種別（有給・欠勤・休暇）」や「打刻理由」「打刻修正」などを入れる場合、`attendances` に `status` や `approval` フィールドを追加する拡張設計を推奨
- 複数拠点やタイムゾーン対応が必要な場合は `date`/`time` の扱いを UTC ベースに統一し、ゾーン変換を行う

付録：運用・改善提案（短く）

- 本番化のための最小改修
    1. 認証（JWT）導入・HTTPS 強制
    2. DB を PostgreSQL に切替え・接続プール設定
    3. Migrations（Flask-Migrate）でスキーマ管理
    4. API のエラーレスポンスを標準化（JSON: `{code, message, details}`）

5. 入力バリデーションに marshmallow 等を導入してパースとスキーマ検証を明確化

6. ログ（操作ログ・監査ログ）を永続化し、監視アラートを設定

要約（**1**行）

シンプルな REST ベースの勤怠管理 API（Flask + SQLAlchemy + SQLite）で、ユーザー管理・出退勤打刻・月次レポートを提供する。運用に際しては認証、永続 DB、マイグレーション、監視の導入を優先する。