勤怠管理アプリ技術仕様書

目次

プロジェクト概要

1.1 プロジェクト名

勤怠管理アプリ（Attendance Management System）

1.2 技術スタック

- **フロントエンド**: Next.js 14, TypeScript, React
- **バックエンド**: Supabase (PostgreSQL + Auth + Real-time)
- **スタイリング**: CSS Modules
- **テスト**: Jest
- **デプロイ**: Vercel

1.3 主要機能

- スタッフの出退勤記録
- 休憩時間の管理
- 月次勤務時間の集計
- 月跨ぎ勤務の対応
- データ整合性の検証

アーキテクチャ設計

2.1 全体構成

```
src/
├── app/
│   ├── attendance/          # 勤怠管理機能
│   │   ├── page.tsx         # メインページ
│   │   ├── hooks/           # カスタムフック
│   │   ├── types.ts         # 型定義
│   │   ├── utils/           # ユーティリティ
│   │   └── testData.ts      # テストデータ
│   └── utils/
│       └── supabase/        # Supabase設定
├── __tests__/               # テストファイル
└── docs/                    # ドキュメント
```

2.2 レイヤー構造

```
┌─────────────────────────────────────────┐
│           Presentation Layer            │
│          (UI Components)                │
├─────────────────────────────────────────┤
│           Business Logic Layer          │
│          (Custom Hooks)                 │
├─────────────────────────────────────────┤
│           Data Access Layer             │
│          (Supabase Client)              │
├─────────────────────────────────────────┤
│           Database Layer                │
│          (PostgreSQL)                   │
└─────────────────────────────────────────┘
```

データモデル

3.1 データベーススキーマ

3.1.1 staff テーブル

```sql
CREATE TABLE staff (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

3.1.2 attendance テーブル

```sql
CREATE TABLE attendance (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  staff_id UUID REFERENCES staff(id) ON DELETE CASCADE,
  clock_in TIMESTAMP WITH TIME ZONE NOT NULL,
  clock_out TIMESTAMP WITH TIME ZONE,
  break_start TIMESTAMP WITH TIME ZONE,
  break_end TIMESTAMP WITH TIME ZONE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

3.2 TypeScript型定義

3.2.1 基本型定義

```typescript
export type Staff = {
  id: string;
  name: string;
};

export type AttendanceRecord = {
  date: string;                // 表示用日付（YYYY/MM/DD）
  clockIn: string;             // 表示用出勤時間（HH:mm）
  clockOut: string | null;     // 表示用退勤時間（HH:mm）
  isCrossDay: boolean;         // 日付跨ぎフラグ
  originalClockIn: string;     // 計算用出勤時間（ISO）
  originalClockOut: string | null; // 計算用退勤時間（ISO）
  breakStart: string | null;   // 休憩開始時間（ISO）
  breakEnd: string | null;     // 休憩終了時間（ISO）
};
```

コアロジック

4.1 勤務時間計算ロジック

4.1.1 基本勤務時間計算

```typescript
const calculateWorkTime = (clockIn: string, clockOut: string): string => {
  const start = new Date(clockIn);
  const end = new Date(clockOut);

  const diffMinutes = Math.floor((end.getTime() - start.getTime()) / (1000 * 60));

  if (diffMinutes < 0) return '00:00';

  const hours = Math.floor(diffMinutes / 60);
  const mins = diffMinutes % 60;

  return `${hours.toString().padStart(2, '0')}:${mins.toString().padStart(2, '0')}`;
};
```

ロジック説明:

1. ISO形式の時刻文字列をDateオブジェクトに変換
2. ミリ秒単位の差分を計算
3. 分単位に変換（切り捨て）
4. 時間と分に分割してHH:mm形式で返却

4.1.2 月跨ぎ勤務時間計算

```
const calculateWorkTimeForPeriod = (
  clockIn: string,
  clockOut: string,
  periodStart: Date,
  periodEnd: Date
): string => {
  const start = new Date(clockIn);
  const end = new Date(clockOut);

  // 期間内の有効な勤務時間を計算
  const effectiveStart = new Date(Math.max(start.getTime(), periodStart.getTime()));
  const effectiveEnd = new Date(Math.min(end.getTime(), periodEnd.getTime()));

  if (effectiveStart.getTime() >= effectiveEnd.getTime()) {
    return '00:00';
  }

  const diffMinutes = Math.floor((effectiveEnd.getTime() - effectiveStart.getTime()) / (1000 * 60));

  const hours = Math.floor(diffMinutes / 60);
  const mins = diffMinutes % 60;

  return `${hours.toString().padStart(2, '0')}:${mins.toString().padStart(2, '0')}`;
};
```

ロジック説明:

1. 勤務期間と集計期間の重複部分を特定
2. 重複期間内の勤務時間のみを計算
3. 月跨ぎ勤務でも正確な月次集計を実現

4.1.3 休憩時間計算

```
const calculateBreakTime = (breakStart: string | null, breakEnd: string | null): string => {
  if (!breakStart || !breakEnd) return '-';

  const start = new Date(breakStart);
  const end = new Date(breakEnd);
  let diffMinutes = Math.ceil((end.getTime() - start.getTime()) / (1000 * 60));

  if (diffMinutes < 0) diffMinutes = 0;

  const hours = Math.floor(diffMinutes / 60);
  const mins = diffMinutes % 60;

  return `${hours.toString().padStart(2, '0')}:${mins.toString().padStart(2, '0')}`;
};
```

4.1.4 実作業時間計算（休憩時間差し引き）

```
const calculateActualWorkTime = (
  clockIn: string,
  clockOut: string,
  breakStart: string | null,
  breakEnd: string | null
): string => {
  const start = new Date(clockIn);
  const end = new Date(clockOut);

  // 総勤務時間（分）
  let totalMinutes = Math.ceil((end.getTime() - start.getTime()) / (1000 * 60));
  if (totalMinutes < 0) totalMinutes = 0;

  // 休憩時間（分）
  let breakMinutes = 0;
  if (breakStart && breakEnd) {
    const bStart = new Date(breakStart);
    const bEnd = new Date(breakEnd);
    breakMinutes = Math.ceil((bEnd.getTime() - bStart.getTime()) / (1000 * 60));
    if (breakMinutes < 0) breakMinutes = 0;
  }

  // 実作業時間（分）
  let actualMinutes = totalMinutes - breakMinutes;
  if (actualMinutes < 0) actualMinutes = 0;

  const hours = Math.floor(actualMinutes / 60);
  const mins = actualMinutes % 60;

  return `${hours.toString().padStart(2, '0')}:${mins.toString().padStart(2, '0')}`;
};
```

ロジック説明:

1. 総勤務時間を計算（出勤から退勤まで）

2. 休憩時間を計算（休憩開始から終了まで）
3. 総勤務時間から休憩時間を差し引き
4. 負の値にならないよう制御

4.2 データ整合性検証ロジック

4.2.1 勤怠記録の整合性チェック

```
const validateRecords = (records: { clock_in: string; clock_out: string | null }[]) => {
  if (records.length === 0) return true;

  let hasUnclockedOut = false;

  for (let i = 0; i < records.length; i++) {
    const record = records[i];
    if (!record.clock_out) {
      if (hasUnclockedOut) {
        // 複数の未退勤記録がある場合は不正
        return false;
      }
      hasUnclockedOut = true;
    }
  }
  return true;
};
```

検証ルール:

- 未退勤の記録は1つまで許可
- 複数の未退勤記録がある場合は不正

4.3 月次集計ロジック

4.3.1 160時間制限付き月次集計

```
const calculateMonthlyTotal = (records: AttendanceRecord[], year: number, month: number) => {
  let totalMinutes = 0;
  const maxMinutes = 160 * 60; // 160時間制限

  for (const record of records) {
    if (record.clockOut && record.originalClockIn && record.originalClockOut) {
      const start = new Date(record.originalClockIn);
      const end = new Date(record.originalClockOut);
      const monthStart = new Date(year, month, 1, 0, 0, 0, 0);
      const monthEnd = new Date(year, month + 1, 1, 0, 0, 0, 0);

      // 月跨ぎ判定
      if (start.getMonth() !== end.getMonth() || start.getFullYear() !== end.getFullYear()) {
        // 前月分の計算
        if (start.getFullYear() === year && start.getMonth() === month) {
          const midnight = new Date(start);
          midnight.setHours(24, 0, 0, 0);
```

```
        const diffMinutes = Math.ceil((midnight.getTime() - start.getTime()) / (1000 * 60));
        if (totalMinutes + diffMinutes > maxMinutes) {
          totalMinutes = maxMinutes;
          break;
        }
        totalMinutes += diffMinutes;
      }
      // 当月分の計算
      if (end.getFullYear() === year && end.getMonth() === month) {
        const monthStartMidnight = new Date(end);
        monthStartMidnight.setHours(0, 0, 0, 0);
        const diffMinutes = Math.ceil((end.getTime() - monthStartMidnight.getTime()) / (1000 * 60));
        if (totalMinutes + diffMinutes > maxMinutes) {
          totalMinutes = maxMinutes;
          break;
        }
        totalMinutes += diffMinutes;
      }
    } else {
      // 通常勤務
      const effectiveStart = new Date(Math.max(start.getTime(), monthStart.getTime()));
      const effectiveEnd = new Date(Math.min(end.getTime(), monthEnd.getTime()));
      if (effectiveStart.getTime() < effectiveEnd.getTime()) {
        const diffMinutes = Math.ceil((effectiveEnd.getTime() - effectiveStart.getTime()) / (1000 * 60));
        if (totalMinutes + diffMinutes > maxMinutes) {
          totalMinutes = maxMinutes;
          break;
        }
        totalMinutes += diffMinutes;
      }
    }
  }
}

const hours = Math.floor(totalMinutes / 60);
const minutes = totalMinutes % 60;
return `${hours.toString().padStart(2, '0')}:${minutes.toString().padStart(2, '0')}`;
};
```

ロジック説明:

1. 月跨ぎ勤務を前月分と当月分に分割
2. 各月の期間内の勤務時間のみを集計
3. 160時間制限を適用（超過時は160時間で打ち切り）
4. 分単位で切り上げ処理

UI/UX設計

5.1 コンポーネント構造

5.1.1 メインコンポーネント

```
function AttendanceContent() {
  const {
    staff, records, workTime, status, error,
    handleAttendance, handleBreak
  } = useAttendance(staffId);

  return (
    <div className={styles.container}>
      <Header />
      <StatusDisplay />
      <WorkTimeDisplay />
      <RecordsTable />
      <ActionButtons />
    </div>
  );
}
```

5.1.2 カスタムフック（useAttendance）

```
export const useAttendance = (staffId: string | null) => {
  const [staff, setStaff] = useState<Staff | null>(null);
  const [records, setRecords] = useState<AttendanceRecord[]>([]);
  const [status, setStatus] = useState<AttendanceStatus>({...});

  const fetchData = useCallback(async () => {
    // データ取得ロジック
  }, [staffId]);

  const handleAttendance = async (type: '出勤' | '退勤') => {
    // 出退勤処理ロジック
  };

  const handleBreak = async (type: '休憩開始' | '休憩終了') => {
    // 休憩処理ロジック
  };

  return {
    staff, records, status, error,
    handleAttendance, handleBreak
  };
};
```

5.2 データフロー

```
1. ページロード
  ↓
2. useAttendanceフック実行
  ↓
3. Supabaseからデータ取得
  ↓
4. データ整形・計算
  ↓
5. UI更新
  ↓
6. ユーザー操作
  ↓
7. データベース更新
  ↓
8. 再取得・再表示
```

セキュリティ

6.1 認証・認可

- Supabase Authを使用
- Row Level Security (RLS) でデータアクセス制御
- スタッフは自分のデータのみアクセス可能

6.2 データ検証

- フロントエンド・バックエンド両方でバリデーション
- 時刻の整合性チェック
- 不正な操作の防止

6.3 エラーハンドリング

```
try {
  // データベース操作
} catch (error) {
  console.error('エラー詳細:', error);
  setError(error instanceof Error ? error.message : 'エラーが発生しました');
}
```

テスト戦略

7.1 テスト構造

```
__tests__/
├── attendance.test.ts          # 勤怠ロジックテスト
├── calculations.test.ts        # 計算ロジックテスト
└── components.test.ts          # コンポーネントテスト
```

7.2 テストケース例

7.2.1 勤務時間計算テスト

```
describe('calculateWorkTime', () => {
  test('通常の勤務時間を正しく計算する', () => {
    const result = calculateWorkTime(
      '2024-01-01T09:00:00Z',
      '2024-01-01T18:00:00Z'
    );
    expect(result).toBe('09:00');
  });

  test('月跨ぎ勤務を正しく計算する', () => {
    const result = calculateWorkTimeForPeriod(
      '2024-01-31T23:00:00Z',
      '2024-02-01T07:00:00Z',
      new Date('2024-02-01T00:00:00Z'),
      new Date('2024-02-29T23:59:59Z')
    );
    expect(result).toBe('07:00');
  });
});
```

7.2.2 データ整合性テスト

```
describe('validateRecords', () => {
  test('正常な記録を検証する', () => {
    const records = [
      { clock_in: '2024-01-01T09:00:00Z', clock_out: '2024-01-01T18:00:00Z' }
    ];
    expect(validateRecords(records)).toBe(true);
  });

  test('複数の未退勤記録を検出する', () => {
    const records = [
      { clock_in: '2024-01-01T09:00:00Z', clock_out: null },
      { clock_in: '2024-01-02T09:00:00Z', clock_out: null }
    ];
    expect(validateRecords(records)).toBe(false);
  });
});
```

運用・保守

8.1 ログ管理

- エラーログの記録
- 操作ログの保存
- パフォーマンス監視

8.2 データバックアップ

- Supabaseの自動バックアップ機能
- 定期的なデータエクスポート

8.3 パフォーマンス最適化

- データベースクエリの最適化
- フロントエンドのキャッシュ戦略
- 画像・アセットの最適化

8.4 監視・アラート

- エラー率の監視
- レスポンス時間の監視
- データベース接続の監視

サンプル出力

9.1 勤怠記録テーブル例

日付	開始	終了	休憩	作業時間
01/15（月）	09:00	18:00	01:00	08:00
01/16（火）	08:30	17:30	00:30	08:30
01/17（水）	09:15	18:15	01:15	07:45

9.2 月次集計例

2024年1月合計勤務時間: 160:00

9.3 エラーメッセージ例

- "既に勤務中の記録があります。"
- "出勤記録がありません、または既に退勤済みです。"
- "記録に不整合があります。管理者に連絡してください。"

技術的考慮事項

10.1 時刻処理

- タイムゾーンの統一（UTC使用）
- 夏時間の考慮
- 日付跨ぎの正確な処理

10.2 データ整合性

- トランザクション管理
- 楽観的ロック
- 重複データの防止

10.3 スケーラビリティ

- データベースインデックスの最適化
- クエリの効率化
- キャッシュ戦略

今後の拡張予定

11.1 機能拡張

- 残業時間の自動計算
- 有給休暇の管理
- シフト管理機能

11.2 技術的改善

- リアルタイム更新機能
- モバイルアプリ対応
- API の外部公開

このドキュメントは技術仕様書として作成され、開発・保守・運用の参考資料として使用されます。