

勤怠管理システム プログラム設計書

第1章：システム概要

システムの目的・背景

ここでは本システムの目的と開発の背景について説明します。勤怠管理システムは、従業員の出退勤状況や休暇情報を正確に記録・管理するためのWebベースの社内システムです。紙やExcelでの手動管理によるミスや集計作業の負担を軽減し、従業員と管理者双方の業務効率を向上させることが目的です。また、勤務時間の適正な把握により労務コンプライアンスの強化も図ります。

対象ユーザーと利用シーン

本システムの主な利用者は一般従業員と部門責任者（管理者）です。

- **一般従業員**：出勤時と退勤時にPCやスマートフォンから打刻（出退勤時刻の記録）を行い、自身の勤務実績（日々の勤務時間や月間の総労働時間）を確認します。また、休暇を取得したい際にはシステム上で休暇申請を行います。
- **部門責任者（管理者）**：自部署の従業員の勤怠状況を把握し、承認権限を持ちます。部下から上がってきた休暇申請の内容をシステム上で確認し、承認または否認します。さらに、部門内の勤務実績のレポートをCSV形式で出力したり、新入社員や異動者のユーザーアカウントを登録・管理する役割も担います。

利用シーンとしては、従業員が毎日出勤時に出勤打刻を行い、退社時に退勤打刻を行うのが基本的な流れです。月末や月初には従業員が自身の月次勤務集計を確認し、必要に応じて部門責任者がその内容をチェックします。休暇の申請・承認は随時行われ、承認結果は従業員に通知されます。本システムは小規模～中規模企業での利用を想定しており、同時に利用するユーザー数は数十人程度、全社員数は数百名規模を目安としています。

システム全体構成図

以下に、本システムの全体構成をテキスト図で示します。勤怠管理システムは典型的な3層構造で構築されており、ユーザーのWebブラウザからのアクセスをWebサーバ（ASP.NET Core MVCアプリケーション）が受け取り、データの保存と取得はSQL Serverデータベースで行います。

```
[利用者のPC/スマートフォン] --(HTTPS)--> [Webサーバ (ASP.NET Core MVC)]
                                   ↓ (ODBC/ADO.NET)
                                   [SQL Server データベース]
```

- ・**ユーザー側（プレゼンテーション層）**：従業員や管理者はPCもしくはスマートフォンのWebブラウザを通じてシステムにアクセスします。通信はHTTPSで暗号化され、安全に社内ネットワークまたはクラウド上のサーバに送信されます。
- ・**アプリケーションサーバ側（ビジネスロジック層）**：ASP.NET Core MVCフレームワーク上で動作するWebアプリケーションです。コントローラがユーザーからのリクエストを受け取り、業務ロジックを実行します。業務ロジックでは出退勤打刻の処理や集計計算、申請内容の検証などを行います。

- ・**データベースサーバ側（データ層）**：勤怠情報やユーザー情報を蓄積するSQL Serverデータベースです。アプリケーションからはエンティティフレームワーク（ORM）またはADO.NETを通じてデータの読み書きを行います。

以上がシステム全体の構成となります。社内システムとしてシンプルな構成ですが、HTTPS通信の採用や3層の分離により、セキュリティとメンテナンス性を確保しています。

まとめ

本章では勤怠管理システムの全体像と目的について概説しました。小中規模企業を対象に、従業員と管理者の双方にとって使いやすい勤怠管理の仕組みを提供することが本システムのねらいです。Webブラウザ経由の利用と3層アーキテクチャにより、ユーザーは手軽に利用でき、システム管理者は安全かつ拡張性の高い形でシステムを運用できます。

第2章：開発環境・使用技術

開発言語・フレームワーク・データベース・ツール

本システムの開発に用いるプログラミング言語や主要技術、および使用するツール類を以下に示します。

- ・**開発言語**: C# 8.0（.NET 8.0 ランタイム）
- ・**フレームワーク**: ASP.NET Core MVC (Model-View-Controller)
- ・**データベース**: Microsoft SQL Server (バージョン2019以降を想定)
- ・**ORMツール**: Entity Framework Core（データベースアクセスとO/Rマッピングに使用）
- ・**フロントエンド**: HTML5, CSS3, JavaScript（必要に応じてBootstrap等のCSSフレームワークやjQueryを利用）
- ・**開発ツール**: Visual Studio 2022 / Visual Studio Code（統合開発環境として使用）
- ・**ビルド/デプロイ**: .NET CLI および GitHub Actions / Azure DevOps（CI/CDパイプラインとして）
- ・**バージョン管理**: Git（リモートリポジトリとしてGitHubまたは社内Gitサーバを利用）
- ・**プロジェクト管理**: Azure Boards / JIRA（タスク管理や進捗管理に利用）
- ・**開発/テスト環境**: 開発者PC上にローカルのIIS ExpressとSQL Server Developer版をセットアップ。結合テストや検証用に社内ネットワーク上にテストサーバを用意し、テスト用データベースを構築。

環境構築と実行基盤

- ・開発は主にWindows環境上で行います（Visual Studioを利用）が、ASP.NET Coreはクロスプラットフォーム対応のためLinuxベースのDockerコンテナ上でも動作可能です。将来的にコンテナ技術を用いたデプロイも検討可能です。
- ・本番環境ではWindows Server上にIISを用いてASP.NET Coreアプリをホストするか、またはKestrelを用いた自己ホストを行い、リバースプロキシ（例:Nginx）を経由して公開します。データベースは社内サーバ上のSQL Serverを使用し、必要に応じてAzure SQL Database等のクラウドDBも利用可能です。
- ・バージョン管理はGitフローに則り、`main` ブランチに安定版、`develop` ブランチで開発版を管理します。機能追加や修正はトピックブランチで行い、プルリクエストを経て`develop`へマージします。定期的にリリースブランチを切り、本番環境へデプロイします。

外部ライブラリ・APIの利用

- ・**認証基盤**: ASP.NET Identityを導入し、ユーザー認証やロール管理を実装します。Identityフレームワークによりパスワードハッシュ化やログイン試行制限などのセキュリティ機能を強化します。
- ・**フロントエンドライブラリ**: UIの利便性向上のため、一部画面でJavaScriptライブラリを利用します。例えば日付入力にはカレンダーピッカー（例: flatpickr や jQuery UI DatePicker）を導入し、日付選択

の入力ミスを減らします。また、スタイル統一のためにBootstrap 5を用いてレスポンシブ対応し、PC・モバイルいずれでも見やすい画面とします。

- **CSV出力:** レポート出力ではCSV形式のファイルを生成しますが、これには、.NET組み込みのCSV出力機能（例えばCSVヘルパークラス）を使用します。小規模なデータであるため、外部の報告ツールを使わず簡易に実装します。
- **メール送信API（将来的な利用）:** 現時点で必須ではありませんが、休暇申請の承認結果通知などにメールを活用することを見据え、必要に応じて社内SMTPサーバやSendGrid等のメール配信APIとの連携を検討します。
- **ログ/モニタリング:** ログ出力にはSerilogなどのサードパーティライブラリを使用し、構造化ログをファイルやコンソールに出力します。また、システム監視には将来的にAzure Application InsightsやPrometheusとGrafanaの組み合わせを導入し、アプリケーションのパフォーマンスやエラーレートを計測できるようにします。

まとめ

本章では、システム開発に用いる技術スタックと環境について整理しました。C#とASP.NET Core MVCにより堅牢なWebアプリケーションを構築し、SQL Serverにデータを永続化します。Gitによるバージョン管理やCI/CDの導入で開発効率と品質を高め、必要に応じて外部ライブラリやサービスを活用して機能性と利便性を向上させます。これらの技術基盤により、開発から運用まで一貫した環境を整えています。

第3章：アーキテクチャ設計

採用アーキテクチャパターン

本システムでは、ASP.NET Core MVCフレームワークの特性を活かしつつ、典型的なレイヤードアーキテクチャを採用します。具体的には、プレゼンテーション層（UI/ビューとコントローラ）、ビジネスロジック層（サービスやドメインロジック）、データアクセス層（リポジトリやデータマッパー）に責務を分離した構造とします【図:下記データフロー参照】。

MVC(Model-View-Controller)パターンを基本とし、コントローラはユーザーからの入力を受け取り、モデル（ViewModelやドメインモデル）にデータを詰めてビューへ渡します。モデルとビューの結合度を下げ、可能な限りロジックはビジネス層にカプセル化することで、保守性とテスト容易性を高めています。

各レイヤの責務

- **プレゼンテーション層（UI層）:** ユーザーとの対話を担当します。具体的にはASP.NET Core MVCのコントローラとRazorビューが該当します。コントローラはHTTPリクエストを受け付け、必要なデータをビジネス層から取得してビューに渡します。ビューはそのデータをHTMLとしてレンダリングし、ユーザーに表示します。この層ではできるだけ表示に関する処理と簡単な入力検証のみに留め、ビジネスルールに関わる判断は下位層に委譲します。
- **ビジネスロジック層（ドメイン層）:** 業務上のルールやデータ加工処理を担います。具体的には「勤怠打刻する際の各種チェック処理（同日に二重打刻していないか等）」「月次勤務時間の集計計算」「休暇申請の受付可否判定（過去日付は不可等）」といったロジックを実装します。この層はコントローラから呼び出され、必要に応じてデータ層と連携します。サービスクラスやドメインモデルを配置し、複雑な処理を見通し良くまとめます。また、この層で業務データの整合性を確保し、トランザクション制御（複数のデータ更新がある場合の一括コミット/ロールバック）も担います。
- **データアクセス層（インフラ層）:** データベースとのやり取りを抽象化します。リポジトリパターンを用いて、ビジネス層が直接SQLを意識せずにデータ操作できるようにします。Entity Framework Coreを使用する場合、コンテキストクラスおよびエンティティクラスがこの層に属します。この層では「ユーザーをIDで検索する」「特定月の勤怠記録一覧を取得する」「休暇申請レコードを登録する」といった操作をメソッドとして提供します。データアクセス時の例外処理や接続管理、SQLインジェクション対策（パラメータ利用）などもこの層で行います。

この3層構造により、各層は明確な役割分担を持ち、UI変更やデータベース変更の影響がそれぞれの層に閉じるようになります（例えばUIデザイン変更時はプレゼンテーション層のみ修正、DBの変更はデータ層の修正で吸収可能）。

ディレクトリ構成例

本プロジェクトのコードは役割ごとにディレクトリ分割して整理します。一例として、ソリューションおよびプロジェクトのディレクトリ構成を以下に示します。

```
勤怠管理システム (Solution)
├── AttendanceSystem.Web/      --- プレゼンテーション層 (ASP.NET MVCプロジェクト)
│   ├── Controllers/          --- コントローラクラス
│   ├── Views/                --- Razorビュー (画面ごとのフォルダ配下に.cshtmlファイル)
│   ├── Models/               --- ビューで利用するViewModelやバインド用モデル
│   ├── wwwroot/              --- 静的ファイル (CSS, JavaScript, 画像等)
│   └── Program.cs, Startup.cs --- アプリケーション起動設定 (ミドルウェア、DIコンテナ設定等)
├── AttendanceSystem.Core/     --- ビジネスロジック層 (ドメインモデル・サービスクラス)
│   ├── Services/             --- 業務ロジックを担うサービスクラス
│   └── DomainModels/         --- 業務で扱うドメインオブジェクト (エンティティ、値オブジェクト等)
│       ├── Validators/       --- 入力検証ロジック (バリデーションルール定義等)
├── AttendanceSystem.Infrastructure/ --- データアクセス層 (リポジトリ実装・DB接続)
│   ├── Repositories/         --- リポジトリクラス (インターフェースと実装)
│   └── Entities/             --- データベースのテーブルに対応するエンティティクラス (ORM)
│       └── DbContext.cs      --- Entity Framework用のデータベースコンテキスト
└── AttendanceSystem.Tests/    --- テストプロジェクト
    ├── UnitTests/            --- 単体テスト
    └── IntegrationTests/      --- 結合テスト・シナリオテスト
```

上記のように、プロジェクトをレイヤごとに分割することで、依存関係を一方向に保ちつつ開発を進めます。例えば、CoreやInfrastructureはWebに依存しないクラスライブラリとし、将来的にUIフレームワークを変更してもビジネスロジックを再利用できる設計とします。

データフロー図

次に、本システム内でのデータの流れをテキストで示します。以下は「従業員が出勤打刻を行い、その記録がデータベースに保存される」ケースのデータフロー例です。

```
[ユーザー操作] -- (出勤ボタン押下) --> [コントローラ] --> [サービス層で業務処理] --> [リポジトリ/DB操作] --> [SQL Server]
                                     ← (結果:打刻成功) ← (勤怠記録オブジェクト更新) ←
                                     ← (出勤時刻を保存) ←
←
```

1. **ユーザー操作:** 従業員がWeb画面上で「出勤」ボタンをクリックします。この操作がHTTPリクエストとしてサーバに送信されます。

2. **コントローラ:** リクエストを受け取ったAttendanceController（例）内のPunchInアクションが起動します。ここで基本的な入力チェック（例: 重複打刻防止のため本日既に出勤打刻済みか）を行い、ビジネスロジック層に処理を委譲します。
3. **サービス層:** コントローラから呼ばれたAttendanceService内で、打刻処理のビジネスルールを実行します。例えば、現在時刻を取得し、打刻記録エンティティを生成、休憩時間等の規定があれば付与、さらにデータアクセス層のリポジトリを呼び出します。
4. **データアクセス層:** リポジトリ（AttendanceRepositoryなど）が呼ばれ、内部でEntity Frameworkのコンテキストを使ってAttendance（勤怠記録）テーブルに新規レコードを書き込みます。この際、トランザクションが開始され、コミットにより確実にデータが保存されます。
5. **レスポンス生成:** データ保存が成功すると、サービス層からコントローラに処理結果（例: 打刻レコードオブジェクト）が返されます。コントローラはその結果に応じてビューを選択し、ユーザーに「打刻が正常に完了した」旨を画面表示します。ユーザーには新しい勤怠状態（例えば出勤時刻の表示）が反映された画面が戻り、一連の処理が完了します。

以上のように、ユーザーからの入力は順に各層を通過し、最終的にデータベースに反映されます。逆にデータベースから取得した情報も各層を経てユーザー画面に届けられます。この流れを明確に分離することで、機能追加や変更時に影響範囲を局所化でき、システム全体の安定性と保守性が向上します。

まとめ

アーキテクチャ設計ではMVCにもとづく明確な層分けを行い、役割に応じた分担で開発と保守を容易にしています。プレゼンテーション、ビジネスロジック、データアクセスを切り離すことで、変更に強く拡張性の高い構成となっています。また、データフローを通じて各コンポーネントの連携を示し、システムがどのようにユーザー要求を処理するかを明らかにしました。このようなアーキテクチャにより、将来的な機能拡張や他システムとの連携にも柔軟に対応できる基盤が構築されています。

第4章：画面設計

主要画面一覧

本システムにおける主要な画面とその概要を以下に列挙します。

1. **ログイン画面:** ユーザーID（またはメールアドレス）とパスワードを入力し、本システムにログインする画面です。認証に成功すると出退勤打刻画面（メイン画面）に遷移します。入力フィールドとしてユーザーIDとパスワード、操作として「ログイン」ボタンがあります。不正なログイン時にはエラーメッセージが表示されます。
2. **出退勤打刻画面（メイン）:** 従業員が出勤および退勤の時刻を記録（打刻）するための画面です。現在日時が表示され、ユーザーは「出勤」「退勤」ボタンをクリックすることで打刻できます。また当日の打刻履歴（例: 出勤時刻、退勤時刻、勤務時間の計算結果）が画面上部または別ウィンドウで確認できます。まだ出勤前であれば退勤ボタンは無効化される等、状況に応じたUI制御があります。
3. **日次勤務一覧画面:** ユーザー自身の勤怠記録を日ごとに一覧表示する画面です。通常当月もしくは指定した期間の各日の出勤時刻、退勤時刻、総労働時間、休憩時間（該当する場合）などが表形式で表示されます。従業員はこの画面で自身の過去の勤務実績を確認できます。管理者は部下のユーザーを選択して同様の一覧を確認することもできます。
4. **月次集計画面:** 月単位の勤務時間集計結果を表示する画面です。従業員は選択した月の総労働時間、残業時間、休日勤務時間、休暇日数などの集計値を確認できます。必要に応じて、月次報告書としてCSV形式でダウンロードする機能（管理者のみ）や印刷機能があります。管理者は自部署内の全従業員分の月次集計を一覧または個別に確認できます。
5. **休暇申請画面:** 従業員が休暇（有給休暇、特別休暇等）を申請する画面です。申請フォームとして、休暇の種類、開始日、終了日（連続した休暇の場合）、理由の入力欄があります。ユーザーはこれらを

入力し「申請送信」ボタンを押して休暇申請を送信します。送信後、申請は承認待ち状態となり、承認者の確認待ちとなります。

6. **休暇承認画面:** 部門責任者（管理者）が部下からの休暇申請を確認し、承認もしくは却下の処理を行う画面です。未承認の申請一覧がテーブル形式で表示され、各申請ごとに詳細（申請者名、休暇期間、理由）を閲覧できます。承認者は各行に用意された「承認」または「否認」ボタンをクリックし、必要に応じてコメントを入力して処理します。処理結果はデータベースに反映され、申請者に通知されます。
7. **管理者メニュー（ユーザー管理画面、レポート出力画面）:** 管理者権限ユーザーのみがアクセスできるメニューです。ユーザー管理画面では従業員アカウントの一覧表示、新規登録、情報編集、退職者のアカウント無効化などを行います。レポート出力画面では指定した部門・期間の勤怠データをCSVダウンロードする機能を提供します。例えば、部門責任者は自部署の当月の勤怠集計をCSVで出力し、給与計算や労務管理に利用できます。

各画面の構成要素

各画面に含まれる主なUI要素（入力項目やボタン、表示内容）を簡潔に整理します。

- **ログイン画面:** ユーザーID入力フィールド、パスワード入力フィールド、ログインボタン、（必要に応じて「パスワードを忘れた場合」リンク）、エラーメッセージ表示領域。シンプルな画面構成で、中央にログインフォームを配置します。
- **出退勤打刻画面:** 現在日時の表示ラベル、当日の打刻状況表示（「未出勤」もしくは「出勤時刻：〇〇、退勤時刻：--」などのステータス）、「出勤」ボタン（出勤前のみ活性）、「退勤」ボタン（出勤打刻後に活性化）。画面上部やサイドにメニュー（勤怠一覧、休暇申請、ログアウト等へのリンク）を配置します。
- **日次勤務一覧画面:** 期間選択ドロップダウン（またはカレンダー）や検索フィルター、勤怠一覧表（列項目：日付、曜日、出勤時刻、退勤時刻、総労働時間、備考など）。各行が1日分の記録を表し、ユーザーごとまたは日付順に並び替え可能にします。画面上部には年月の切り替えナビゲーション、管理者にはユーザー選択プルダウンが追加表示されます。
- **月次集計画面:** 月度選択プルダウン、集計結果の表示領域（総労働時間、残業時間、休暇日数等のサマリーをカード状または表形式で表示）。必要に応じて詳細リンク（日次一覧への遷移）や、CSVダウンロードボタン（管理者向け）が配置されます。画面デザインはシンプルに、重要な数値を大きく表示するなど視認性を高めます。
- **休暇申請画面:** 休暇種類選択ドロップダウン（例：「有給休暇」「特別休暇」等）、開始日入力（カレンダーUI付き）、終了日入力（1日休暇の場合は開始日と同じ）、理由テキストエリア、申請送信ボタン。入力必須項目にはラベルに「*」を付与し、未入力時はエラーメッセージをフォーム内に表示します。送信後は「申請を受け付けました」の通知を表示します。
- **休暇承認画面:** 未処理の休暇申請一覧表（列項目：申請ID、申請者氏名、休暇期間、休暇種類、申請日時、理由、状態）。各行の末尾に「承認」「否認」ボタンがあり、クリックすると承認用のモーダルダイアログが開き、コメント入力欄および最終確認ボタンが表示されます。承認操作後、その申請行は一覧から消え、画面上部に処理完了のメッセージが表示されます。
- **ユーザー管理画面:** ユーザー一覧表（氏名、ユーザーID、部署、役職、アカウント状態（有効/無効）等の列）、新規ユーザー登録ボタン、各ユーザー行に編集ボタンや無効化ボタン。新規登録や編集の際は別画面またはモーダルでユーザー情報（氏名、メール、ロール等）を入力するフォームを表示します。ユーザー数が多い場合のために検索バーやページネーションも用意します。
- **レポート出力画面:** レポート条件指定フォーム（例：対象年月または期間、部署選択プルダウン）、および「CSVダウンロード」ボタン。画面自体には結果は表示せず、ボタン押下で即座にCSVファイルのダウンロードが開始されます。ダウンロード完了後、画面上部に「レポートをCSVに出力しました」といった簡単な通知を表示します。

画面遷移図

主要な画面間の遷移を簡易なテキスト図で表します。ユーザー（一般従業員）と管理者それぞれの遷移を示しています。

未ログイン状態:

[ログイン画面] -- (認証成功) --> [出退勤打刻(メイン画面)]
(認証失敗) --> [ログイン画面 (エラー表示)]

一般従業員のメイン画面から:

[出退勤打刻画面] --> [日次勤務一覧画面]
 \--> [月次集計画面]
 \--> [休暇申請画面]
 \--> [ログアウト -> ログイン画面]

管理者メニュー:

[出退勤打刻画面] --> [ユーザー管理画面]
 \--> [レポート出力画面]

休暇申請フロー:

[休暇申請画面] -- (申請送信) --> [出退勤打刻画面 (申請受付メッセージ表示)]
[休暇承認画面] <-- (管理者がメニューからアクセス) -- [出退勤打刻画面]

承認処理:

[休暇承認画面] -- (承認/否認実行) --> [休暇承認画面 (一覧更新・完了通知)]
(処理完了後) --> [出退勤打刻画面または管理者メニュー]

- ・ログイン後、一般従業員はまず出退勤打刻画面に遷移します。そこから自分の勤怠一覧や集計、休暇申請画面へ移動できます。
- ・管理者は出退勤打刻画面に加えて管理者メニュー（ユーザー管理、レポート出力）への遷移リンクが表示されます。
- ・休暇申請画面で申請を送信するとメイン画面に戻り、申請が受付状態になる流れです。管理者は別途休暇承認画面で保留中の申請を処理できます。
- ・ログアウトするとログイン画面に戻り、再度認証しない限り他の画面にはアクセスできません。

主要UIのワイヤーフレーム例

主要な画面について、簡易なテキストベースのワイヤーフレームを示します。レイアウトのイメージや要素配置の参考とします。

ログイン画面ワイヤーフレーム:

```
+-----+
|  勤怠管理システム ログイン  |
+-----+
| ユーザーID [_____] |
| パスワード [_____] |
|                   |
| [ ログイン ] (ボタン) |
|                   |
+-----+
```

```

|                                     |
| ※ユーザーIDとパスワードを入力し |
| ログインしてください。           |
|                                     |
| <エラーメッセージエリア>         |
+-----+

```

図: ログイン画面ではユーザーIDとパスワードの入力欄、およびログインボタンが中央に配置される。エラー時にはフォーム下部にメッセージを表示。

出退勤打刻画面ワイヤーフレーム:

```

+-----+
| [ユーザー名] さんの勤怠状況      (ログアウト) |
+-----+
| 日時: 2025/10/30 08:59:58 現在          |
|                                           |
| 状態: 未出勤 (本日まだ出勤打刻がありません) |
|                                           |
| [ 出勤打刻 ] (ボタン)   [ 退勤打刻 ] (ボタン) |
|                                           |
+-----+
| ◎本日の記録                          |
|   出勤時刻 --:--   退勤時刻 --:--   |
|   備考: 未打刻                        |
|                                           |
| メニュー: [日次一覧] [月次集計] [休暇申請] |
|           (管理者のみ)[ユーザー管理][レポート] |
+-----+

```

図: 出退勤打刻画面。上部に現在日時と勤務状態を表示し、打刻ボタンを配置。打刻後は本日の記録欄に時刻が反映される。画面下部またはサイドにメニューリンクを配置。管理者には追加メニューが見える。

休暇申請画面ワイヤーフレーム:

```

+-----+
| 休暇申請                             | [戻る] |
+-----+
| 種類: [有給休暇 v]                   |
| 開始日: [2025/11/10] (カレンダー)    |
| 終了日: [2025/11/12] (カレンダー)    |
| 理由: [ 3日間の私用のため           ] |
|           [ ]                         |
|                                           |
| [ 申請送信 ] (ボタン)                 |
+-----+
| 注: 有給残日数: 5日 (参考情報表示)   |
+-----+

```

図: 休暇申請画面。休暇種別、期間、理由を入力するフォームを表示。送信ボタン押下で申請処理が実行される。必要に応じて残有給日数などの参考情報も表示。

まとめ

画面設計では、従業員と管理者が直感的に操作できるようシンプルで分かりやすいUIを目指しました。各画面の要素配置や遷移を明確にし、日常的な勤怠打刻や休暇申請がスムーズに行えるデザインとしています。ワイヤーフレームによるレイアウトイメージも示し、実装前に画面の完成像を共有することで、認識齟齬を防ぎます。これらの設計により、ユーザーエクスペリエンスの高い使いやすいシステム画面を提供します。

第5章：機能設計

各機能の処理フロー

本システムに実装される主要な機能について、入力から処理、出力までのフローを示します。ユーザー操作を起点に、システム内部でどのような処理が行われ、最終的に何をユーザーに返すかを整理します。

ログイン機能

- ・**入力:** ユーザーID、パスワード（ログイン画面で入力）
- ・**処理:** 入力されたユーザーIDをもとにデータベースからユーザー情報を取得し、保存されているパスワードハッシュと入力パスワードを照合します。認証成功の場合、ユーザーのセッションを開始し、ユーザーのロールに応じて適切なトップページ（通常は出退勤打刻画面）へ遷移します。認証失敗の場合、エラーメッセージを生成します。また一定回数以上認証に失敗した場合はアカウントをロックします（一定時間ログイン不可）。
- ・**出力:** 認証成功時は出退勤打刻画面へのリダイレクト（ユーザーには勤怠メイン画面が表示される）。失敗時はログイン画面にエラーメッセージ（「ユーザーIDまたはパスワードが正しくありません」等）を表示します。

ログアウト機能

- ・**入力:** （ユーザー操作なし、ログアウトリンクのクリック）
- ・**処理:** サーバ側でユーザーのセッション情報を無効化（削除）し、認証クッキーを破棄します。
- ・**出力:** ログイン画面へのリダイレクト。これにより、ユーザーはシステムから安全にログアウトされます。

出退勤打刻機能

- ・**入力:** 現在ログインしているユーザー（従業員）の操作（「出勤」または「退勤」ボタン押下）
- ・**処理:** システム時刻とユーザーIDを取得し、当該ユーザーの当日の勤怠記録を処理します。
- ・「出勤」ボタン押下時: 処理開始時点で当日の出勤記録が既に存在しないかを確認します（同一ユーザー・同日で未退勤の記録がないことを確認）。問題なければ現在時刻で新規の勤怠レコードを作成し、出勤時刻を記録します。既に出勤済みの場合は二重打刻エラーとします。
- ・「退勤」ボタン押下時: 当日の勤怠記録に対応する未退勤（退勤時刻が未設定）のレコードを検索します。見つければ現在時刻を退勤時刻として記録し、勤務時間（退勤時刻-出勤時刻から休憩時間を差し引いた値など）を計算・更新します。該当レコードが見つからない場合は打刻エラー（「出勤打刻がありません」等）とします。
- ・**出力:** 打刻処理が成功した場合、画面上で当日の勤務状況を更新表示します（例: 「出勤時刻 9:00 登録済み」のメッセージや、退勤時には総労働時間を計算して表示）。また、処理完了メッセージ（「打刻が完了しました」等）を一時的に表示します。エラー時はエラーメッセージを画面上部等に表示し、必要に応じてユーザーに再操作を促します。

日次勤務一覧表示機能

- **入力:** 照会対象期間（デフォルトでは当月）および（管理者の場合）照会対象のユーザー選択
- **処理:** 指定されたユーザー・期間に対し、データベースの勤怠記録（Attendanceテーブル）から該当する日々の出勤・退勤データを抽出します。抽出にはユーザーIDと日付範囲を条件としたクエリを実行します。取得したデータを日付順に整列し、各日について必要な派生情報を計算します（例: 勤務時間＝退勤-出勤-休憩、欠勤日の判定など）。
- **出力:** 日次勤怠の一覧表を画面に表示します。各行には日付、曜日、出勤時刻、退勤時刻、勤務時間等が含まれます。該当期間におけるデータが存在しない場合は「記録がありません」等のメッセージを表示します。また、管理者による他ユーザー閲覧の場合は画面上に対象ユーザー名を明示します。

月次集計表示機能

- **入力:** 照会対象月（例: 2025年10月）およびユーザー（管理者の場合は部署内全員または特定ユーザー選択）
- **処理:** 対象月の勤怠データをデータベースから集計します。具体的には、指定月の全出勤記録を取得し、集計関数で総労働時間、残業時間、遅刻・早退回数、休暇取得日数等を算出します。ビジネスロジック層で月次の計算を行い、法定労働時間との比較やアラート判定（例えば残業時間が一定時間を超えた場合の警告）も実施します。
- **出力:** 計算結果を画面に表示します。従業員には自身の月間サマリー（総労働時間、残業○時間等）が表示され、管理者には選択した集計対象（部署全体またはユーザー）のサマリーが表示されます。また、管理者画面ではCSV出力ボタンが有効になります。

休暇申請機能

- **入力:** 休暇申請フォームの内容（休暇種類、開始日、終了日、理由）、申請操作（送信ボタン押下）
- **処理:** 入力チェックを実施します。必須項目の未入力確認、日付範囲の妥当性確認（終了日は開始日以降か、過去日が含まれていないか）、残り有給日数との比較（有給休暇の場合、残日数以上の申請は不可）などのバリデーションを行います。問題がなければ、休暇申請データを作成しデータベースに挿入します。申請レコードにはステータス（未承認）と申請日時、申請者ID、承認者ID（想定される承認者。部門責任者に紐づけるためユーザーテーブルから自動設定）などを含みます。必要に応じて承認者へ通知（メール送信やシステム内通知）をトリガします。
- **出力:** 申請送信後、ユーザーの画面上に「休暇申請を受け付けました（承認待ち）」といった確認メッセージを表示します。申請フォームの入力内容はクリアされ、ユーザーは引き続き他の操作（画面遷移等）が可能です。エラーがある場合（例: 未入力項目がある、期間重複など）はその旨をメッセージ表示し、申請処理を中断します。

休暇承認機能

- **入力:** 管理者が休暇承認画面で行う操作（各申請に対する「承認」または「否認」ボタンのクリック、および必要に応じてコメント入力）
- **処理:** 管理者がボタンを押下すると対象の申請IDと処理種別（承認/否認）がシステムに送信されます。サーバ側では該当申請レコードを検索し、現在のステータスをチェックします（既に処理済みでないことを確認）。承認の場合はステータスを「承認済」、否認の場合は「否認済」に更新し、承認者IDおよび処理日時、コメント（あれば）を記録します。この際、データの整合性を保つためトランザクション内で更新処理を実行します。併せて、承認/否認の結果を申請者へ通知します（次回ログイン時の通知表示やメール通知など）。
- **出力:** 管理者画面上で該当の申請が一覧から消え、処理結果のメッセージ（「申請ID1234を承認しました」等）を表示します。申請者側には後ほど、自身の休暇申請状況に承認結果が反映されます。例えば、日次一覧や別途用意する申請状況確認画面でステータスが「承認済（承認者名、日時）」と表示されます。否認の場合も同様です。

ユーザー管理機能

- **入力:** 管理者によるユーザー管理画面での操作（例: 「新規ユーザー登録」ボタン押下、ユーザー情報フォーム入力、「保存」押下。または既存ユーザーの「編集」「無効化」操作）
- **処理:**
 - 新規登録の場合: 管理者がフォームに入力したユーザー情報（氏名、メールアドレス等）を受け取り、重複するユーザーID/メールが既に存在しないか検証します。ユニークキー制約に違反しなければ、新しいユーザーアカウントを作成します。パスワードは初期パスワードを自動生成しハッシュ化して保存、または招待メールを送る運用も可能です。ロール（一般/管理者）も設定し、必要なら部署や上長情報も紐付けます。
 - 編集の場合: 選択されたユーザーIDに対し、フォームから送信された更新情報（氏名変更やロール変更等）でデータベースのユーザーテーブルを更新します。権限変更など重要な操作は再確認ダイアログを出すことも検討します。
 - 無効化（退職者処理）の場合: ユーザーテーブルのステータス（有効フラグ）をオフにするか、削除フラグを立てます。勤怠履歴は残す必要があるため物理削除は行わず、論理削除とします。
- **出力:** いずれの操作も成功時には「ユーザーを登録しました」「ユーザー情報を更新しました」「アカウントを無効化しました」等のメッセージを管理者画面上に表示します。ユーザー一覧は最新の情報で再表示され、変更が反映されます。エラー発生時（例: 入力値の不備や重複エラー）は画面上にエラーメッセージを表示し、必要箇所をハイライトします。

レポート出力機能

- **入力:** 管理者がレポート出力画面で指定する条件（例: 部署=〇〇部, 対象期間=2025年10月）および「CSVダウンロード」ボタンのクリック
- **処理:** 指定条件に従い、対象範囲の勤怠データをデータベースから取得します。例えば2025年10月の〇〇部全従業員の勤怠記録を日別集計したデータセットを生成します。取得データをCSV形式（カンマ区切りテキスト）に整形します。1行目にヘッダー（列名）、以降各行に日付、氏名、出勤時刻、退勤時刻、総労働時間、休暇種別（該当日の休暇があれば）等を含めます。データ量が多い場合はメモリに載せすぎないようにストリーム処理でCSVを書き出します。
- **出力:** HTTPレスポンスとしてCSVファイルをバイナリ出力し、ユーザーのブラウザはファイルのダウンロードを開始します。ファイル名は例えば `AttendanceReport_202510_〇〇部.csv` のように日時や部署がわかるよう命名します。ダウンロード完了後、画面には簡単な完了メッセージを表示します。何らかの理由でデータが取得できない場合は、画面上に「レポート出力に失敗しました。条件を見直してください」等のエラーメッセージを表示します。

バリデーションルール

各機能における入力値の検証ルールをまとめます。サーバ側での厳密なチェックに加え、可能な限りクライアント側(Javascript/HTML5)の簡易チェックも活用します。

- **ログイン:** ユーザーID・パスワードはいずれも必須入力です。入力フォーマットは事前に案内（例えばパスワードの最低文字数など）し、未入力時は「ユーザーIDを入力してください」「パスワードを入力してください」と表示します。不正な組み合わせ検出時は一般的なメッセージでまとめ（セキュリティ上、どちらが間違っているか特定させない）ます。
- **出退勤打刻:** 打刻ボタン押下時はサーバ側で二重打刻防止チェックを必ず実施します。また、日付や時刻はサーバ側時刻を採用し、ユーザーのローカル時刻は信用しません。退勤打刻時には対応する出勤が未登録の場合エラーとします。
- **日次一覧/月次集計:** ユーザーからの期間指定入力がある場合、その形式（YYYY-MM-DDなど）を検証し、不正な場合はデフォルト期間にフォールバックします。存在しない日付や未来の期間のみ指定などはエラーとし、「正しい日付を指定してください」と表示します。
- **休暇申請:** 必須項目（種類、開始日、終了日、理由のうち社内規定で必須とするもの）は未入力チェックを行います。開始日と終了日の前後関係が逆転していないか確認します。同一期間で既に申

請済みの休暇がないか（重複申請チェック）を行います。有給休暇の場合、申請日数が残有給数を超えていればエラーにします。入力文字列（理由欄等）は長さ制限（例えば200文字以内）や特殊文字の除去/エスケープを行い、XSSにならないようにします。

- **休暇承認:** 承認時に、対象申請がすでに処理済みでないことをサーバ側で再確認します（楽観的ロック/悲観的ロックの検討）。コメント欄には長さ制限（例: 500文字以内）を設け、不要なHTMLタグ等は除去します。権限チェックも行い、承認権限のないユーザーからのリクエストは拒否します。
- **ユーザー管理:** 新規登録・編集時のバリデーションとして、メールアドレス形式の正当性チェック、パスワード設定/リセット時の強度チェック（英大小文字・数字・記号を含む一定以上の長さ）を行います。氏名等のテキスト項目は長さ・使用可能文字を制限します。ユーザーIDやメールはユニーク制約をサーバ側で再確認し、重複時はエラーとします。
- **レポート出力:** 条件入力（年月や部署選択）に対し、存在しない部署名が来ないように選択式UIにしています。年月も現在日時より未来は指定できないようUI制御します。サーバ側でも再チェックし、不正なリクエストにはエラーレスポンスを返します。

エラーハンドリング方針

本システムでは、ユーザーに対して可能な限り適切かつ有用なエラー情報を提供しつつ、セキュリティやユーザビリティに配慮したエラーハンドリングを行います。

- **入力エラー:** フォームのバリデーションエラーについては、各入力欄の近傍にメッセージを表示し、どの項目を修正すべきか明示します。例えば「開始日は必須です」「終了日は開始日以降の日付を指定してください」等の具体的な指摘を行います。複数エラーがある場合も、それぞれの項目にメッセージを表示します。
- **業務ロジックエラー:** ビジネスルールに反する操作（例: 二重打刻、残有給不足による申請不可等）の場合は、ユーザーに対してわかりやすい日本語メッセージで原因を伝えます。一方で内部的な詳細（例えば計算結果など）は表示せず、必要に応じてログにのみ残します。
- **システムエラー:** 予期しない例外（DB接続エラーやNULL参照エラー等）が発生した場合、ユーザーには包括的なエラー画面を表示します。画面には「システムエラーが発生しました。管理者にお問い合わせください。」などのメッセージを出し、スタックトレース等は表示しません。同時に、開発者向けにはログに詳細なエラー情報（例外メッセージ、トレース、ユーザーID、操作内容）を記録します。これにより、ユーザーには混乱を与えず、管理者や開発者はログから原因を特定できます。
- **エラー画面/通知:** 全般的なエラーについて、共通のエラーページやダイアログを用意し、UIの一貫性を保ちます。HTTP 404（ページ見つかり）や403（アクセス拒否）についてはカスタムエラーページを設定し、適切なナビゲーション（トップへ戻るリンク等）を提供します。メンテナンス時には専用のメンテナンスページを表示し、ユーザーにシステム停止中であることを明示します。
- **例外処理の実装:** アプリケーション内ではtry-catchを適切に用いて例外を捕捉し、必要に応じて再スローします。ASP.NET Coreのミドルウェア（UseExceptionHandler）を活用し、グローバルなエラーハンドラで未処理例外をキャッチしてエラーページへリダイレクトする設計とします。

以上の方針により、ユーザー体験を損ねず、かつ原因究明に必要な情報は保持するバランスの取れたエラーハンドリングを実現します。

代表的な機能の擬似コード例

以下に、代表的な機能である「出退勤打刻処理」の擬似コードを示します。実際のC#構文とは異なりますが、処理の流れと主要なロジックを表現しています。

```
// 出退勤打刻処理 (擬似コード)
function PunchAttendance(userId, action) {
    // 今日の日付を取得
    date = Today();
```

```

if (action == "IN") {
    // 当日の未退勤レコードが既にあるか確認
    record = AttendanceRepository.FindByUserAndDate(userId, date);
    if (record != null && record.OutTime == null) {
        return Error("既に出勤打刻済みです。");
    }
    // 新規勤怠レコードを作成
    newRecord = new AttendanceRecord {
        UserId = userId,
        Date = date,
        InTime = CurrentTimestamp(),
        OutTime = null,
        WorkHours = 0
    };
    AttendanceRepository.Insert(newRecord);
    return Success("出勤打刻が完了しました。", newRecord);
}
else if (action == "OUT") {
    // 当日の未退勤レコードを取得
    record = AttendanceRepository.FindByUserAndDate(userId, date);
    if (record == null || record.OutTime != null) {
        return Error("出勤打刻が見つかりません。");
    }
    // 退勤時刻を記録
    record.OutTime = CurrentTimestamp();
    // 勤務時間を計算 (休憩時間控除は省略)
    record.WorkHours = CalculateWorkHours(record.InTime, record.OutTime);
    AttendanceRepository.Update(record);
    return Success("退勤打刻が完了しました。", record);
}
}

```

上記の擬似コードでは、`PunchAttendance` 関数がユーザーIDとアクション ("IN"または"OUT") を受け取り、出勤打刻時には既存記録の有無チェック、新規レコード挿入を行い、退勤打刻時には当日記録の検索、退勤時間の登録と勤務時間計算を行っています。エラー条件では適切なメッセージを返し、正常時には更新後の勤怠レコードを返しています。実際の実装ではこれに加えてトランザクション管理や時刻の丸め（例えば15分単位丸め等のビジネスルール）などが入りますが、基本的な処理手順はこの通りです。

まとめ

本章では主要機能ごとの処理手順やロジックを詳述しました。入力から出力までのフローを明確にすることで、実装時の見落としを防ぎ、また関係者との認識共有を図ります。擬似コード例により、特に重要な打刻処理の流れを示し、具体的なアルゴリズムの理解を助けています。これらの機能設計をもとに、今後の詳細設計・実装フェーズで適切なコード構造とロジック実装を行っていきます。

第6章：データベース設計

ER図（エンティティ間の関係）

本システムで使用する主要なテーブルとそのリレーション（エンティティ間の関連）を示します。テキストでの簡易ER図を以下に記載します。

```
[Users] 1 ---* [AttendanceRecords]
|         (Users.UserId = AttendanceRecords.UserId)
|
1 ---* [LeaveRequests]
      (Users.UserId = LeaveRequests.UserId)
      (Users.UserId = LeaveRequests.ApproverId)
```

- **Users（ユーザーテーブル）**：従業員および管理者のアカウント情報を保持します。各ユーザーは複数の勤怠記録（AttendanceRecords）および複数の休暇申請（LeaveRequests）を持つことができます。
- **AttendanceRecords（勤怠記録テーブル）**：各従業員の日々の出退勤時刻を記録します。各レコードは1人のユーザーに紐づきます（UserId）。ユーザーが削除（退職）された場合でも勤怠記録は履歴として保持されます。
- **LeaveRequests（休暇申請テーブル）**：従業員の休暇申請情報を記録します。申請者（UserId）と承認者（ApproverId）がそれぞれユーザーテーブルのレコードを参照します（自己結合関係）。1つの申請には単一の承認者が割り当てられます。
- **（Departments（部署テーブル）**：任意 - 将来的拡張）各ユーザーの所属部署情報を管理します。部門責任者とユーザーを部署単位で関連付ける際に利用できます。本設計では必須ではないため省略しますが、中規模以上の組織対応では追加を検討します。

テーブル定義書

各テーブルのスキーマ（カラム構成）を示します。カラム名、データ型、キー制約、概要は以下の通りです。

Usersテーブル

カラム名	データ型	PK	FK	概要 (説明)
UserId	INT (自動採番)	YES	NO	ユーザーID（一意の連番）
UserName	NVARCHAR(100)	NO	NO	ユーザーのログインID（メールアドレス等）※ユニーク制約
PasswordHash	VARBINARY(256)	NO	NO	ハッシュ化済みパスワード
FullName	NVARCHAR(100)	NO	NO	氏名（表示名）
Role	TINYINT	NO	NO	ユーザーの役割種別（0:一般, 1:管理者 等）
ManagerId	INT	NO	YES	上長ユーザーID（FK: Users.UserId 自身へのFK）
IsActive	BIT	NO	NO	アカウント有効フラグ（退職者はFalse）
CreatedAt	DATETIME	NO	NO	アカウント登録日時

カラム名	データ型	PK	FK	概要 (説明)
UpdatedAt	DATETIME	NO	NO	最終更新日時

- **UserName**には一意制約を設け、重複したログインIDが登録されないようにします。
- **PasswordHash**は平文パスワードを安全に保存するためにソルト付きハッシュを格納します。
- **Role**はユーザーの権限を表し、0=一般従業員、1=管理者（部門責任者）など数値で管理します。
- **ManagerId**はこのユーザーの直属上司となるユーザーのIDです。管理者自身やトップレベルの管理者はNULLもしくは自分自身を指す値とします（運用に応じ設定）。
- **IsActive**がFalseのユーザーはログイン不可とし、退職者などの扱いとなります。

AttendanceRecordsテーブル

カラム名	データ型	PK	FK	概要 (説明)
RecordId	INT (自動採番)	YES	NO	勤怠記録ID（ユニークな連番）
UserId	INT	NO	YES (参照: Users.UserId)	対象ユーザーID
WorkDate	DATE	NO	NO	勤務日（年月日）
InTime	DATETIME	NO	NO	出勤時刻
OutTime	DATETIME	NO	NO	退勤時刻（NULL=未退勤）
WorkHours	DECIMAL(5,2)	NO	NO	勤務時間（時間数、小数点以下2桁で時間換算）
Notes	NVARCHAR(200)	NO	NO	備考（例:「遅刻」「早退」等のメモ）
CreatedAt	DATETIME	NO	NO	レコード作成日時
UpdatedAt	DATETIME	NO	NO	レコード更新日時

- **UserId**はUsersテーブルに対する外部キーであり、該当ユーザーが削除されても勤怠履歴は保持するため、Users側削除時はNULLを許容するか論理削除とします（本設計ではユーザー削除は論理削除運用）。
- **WorkDate**は出勤日の年月日部分で、InTime/OutTimeの日付部分と一致します。クエリ効率向上のために持たせています。
- **WorkHours**は勤務時間（時間単位）です。OutTime - InTime から休憩時間を差し引いた値を計算し、小数（例: 8.50時間）で格納します。実運用では分単位ですが、レポート用途で時間を小数表記しています。不要であれば計算時算出でもよいが、集計効率のため保持します。
- **Notes**は任意のメモで、遅刻や早退、欠勤（OutTime無しで退勤打刻忘れ等）の補足を記録します。

LeaveRequestsテーブル

カラム名	データ型	PK	FK	概要 (説明)
RequestId	INT (自動採番)	YES	NO	休暇申請ID（ユニーク連番）
UserId	INT	NO	YES (参照: Users.UserId)	申請者ユーザーID

カラム名	データ型	PK	FK	概要 (説明)
ApproverId	INT	NO	YES (参照: Users.UserId)	承認者ユーザーID (部門責任者)
LeaveType	TINYINT	NO	NO	休暇種別 (1:有給, 2:特休 等コード)
StartDate	DATE	NO	NO	休暇開始日
EndDate	DATE	NO	NO	休暇終了日
Reason	NVARCHAR(200)	NO	NO	申請理由
Status	TINYINT	NO	NO	状態 (0:未承認, 1:承認, 2:否認)
AppliedAt	DATETIME	NO	NO	申請日時
DecidedAt	DATETIME	NO	NO	承認/否認日時 (未承認時はNULL)
Notes	NVARCHAR(200)	NO	NO	承認者からのコメント等 (任意)

- **UserId**および**ApproverId**はいずれもUsersテーブルへのFKです。ApproverIdは申請送信時に自動設定されます (ユーザーのManagerId等から決定)。
- **LeaveType**は休暇の種類コードです。例えば1=年次有給、2=特別休暇、3=病欠等、コード管理します (別途マスターテーブル化も可能)。
- **Status**は申請の現在状態を示します。初期値0 (未承認)、管理者処理で1 (承認) または2 (否認) に更新します。
- **StartDate**と**EndDate**で期間を表現します。1日休暇の場合は同じ日付になります。
- **DecidedAt**は承認者が処理した日時を記録します。未処理の間はNULLのままです。
- **Notes**欄は承認時のコメントなどを残すためのフィールドで、必要に応じて使用します。

テーブル間のリレーション説明

上記のER図およびテーブル定義に示したように、Users - AttendanceRecords 間は1対多、Users - LeaveRequests 間も1対多の関係です。さらにLeaveRequestsでは承認者もUsersテーブルを参照するため、UsersテーブルとLeaveRequestsテーブルの間に2つのリレーションがあります (申請者と承認者)。これにより、あるユーザー (管理者) は複数の申請 (他ユーザーのもの) を承認し得る構造になっています。

外部キー制約として、AttendanceRecords.UserIdはUsers.UserIdに、LeaveRequests.UserIdおよびLeaveRequests.ApproverIdはUsers.UserIdに参照整合性を持ちます。削除や更新時の動作は、ユーザーを削除 (退職) しても関連する勤怠や申請レコードは履歴として残すため、Users削除操作は一般には実行せずIsActiveで論理削除とします。万一物理削除する場合はON DELETE設定でCASCADEではなく、RESTRICT (またはSET NULL) とし、関連レコードが残っているユーザーは削除できない設計とします。

データ取得効率のため、主要な検索に使う列 (UserNameやWorkDate、Statusなど) にはインデックスを適宜付与します。例えばAttendanceRecordsの(UserId, WorkDate)複合インデックスを作成し、特定ユーザーの月次検索を高速化します。

まとめ

データベース設計では、勤怠管理に必要な情報を正規化しつつ、効率的に扱えるスキーマを策定しました。Users・AttendanceRecords・LeaveRequestsといった主要テーブル間の関係を定義し、必要に応じてインデックスや制約を設けています。これにより、データの整合性を保ちながらクエリ性能も確保できます。将来的な要件変更 (例えば部署管理や勤怠パターン追加) にも対応しやすい構造となっており、拡張性の面でも配慮した設計となっています。

第7章：API設計

本システムはASP.NET Core MVCによるサーバサイドレンダリングが基本ですが、一部機能についてはシングルページアプリケーションやモバイルアプリからも利用可能なWeb APIを提供します。以下に、代表的なAPIエンドポイントとその仕様を設計します。

APIエンドポイント一覧

エンドポイント	HTTPメソッド	概要
<code>/api/auth/login</code>	POST	ログイン（認証）し、JWTトークンを取得
<code>/api/auth/logout</code>	POST	ログアウト（サーバ側セッション終了）
<code>/api/attendance/punch/in</code>	POST	出勤打刻（現在ログインユーザーの出勤時刻登録）
<code>/api/attendance/punch/out</code>	POST	退勤打刻（現在ログインユーザーの退勤時刻登録）
<code>/api/attendance/records</code>	GET	勤怠記録一覧取得（クエリパラメータで日付範囲やユーザー指定）
<code>/api/attendance/monthly-summary</code>	GET	月次集計取得（クエリ: 月、ユーザー or 部署指定）
<code>/api/leave/requests</code>	GET	休暇申請一覧取得（自分の申請または承認待ち申請の一覧）
<code>/api/leave/requests</code>	POST	新規休暇申請の登録
<code>/api/leave/requests/{id}/approve</code>	PUT	休暇申請の承認処理
<code>/api/leave/requests/{id}/reject</code>	PUT	休暇申請の否認処理
<code>/api/users</code>	GET	ユーザー一覧取得（管理者用）
<code>/api/users</code>	POST	新規ユーザー登録（管理者用）
<code>/api/users/{id}</code>	PUT	ユーザー情報更新（管理者用）
<code>/api/users/{id}/deactivate</code>	PUT	ユーザー無効化（退職者処理、管理者用）
<code>/api/reports/attendance/monthly</code>	GET	勤怠月次レポートCSV出力（クエリで月や部署指定）

※上記 `/api` 配下のエンドポイントは、必要な認証・認可がなされたうえでアクセス可能となります。例えば `/api/attendance/*` は認証済みユーザーのみ、`/api/users` 系は管理者ロールのみ許可する仕様です。

リクエスト／レスポンス例

以下に、主要なAPIについてリクエストとレスポンスのJSON例を示します。

- ログイン API (`POST /api/auth/login`):
リクエストボディ (JSON) :

```
{
  "username": "tanaka@example.com",
  "password": "Passw0rd!"
}
```

レスポンス（成功時 JSON）：

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...<JWTトークン>",
  "userId": 5,
  "role": "Admin"
}
```

説明: 認証成功時、JWT形式のアクセストークンとユーザーID・ロールを返します。このトークンは以後のAPI呼び出しで `Authorization: Bearer <token>` ヘッダに付与して使用します。認証失敗時は HTTP 401を返し、ボディにエラーメッセージ (`{"error": "Invalid credentials"}`) を含めます。

• 出勤打刻 API (**POST** /api/attendance/punch/in):

リクエストボディ: なし（トークンによりユーザー識別、サーバ側で現在時刻を使用）

レスポンス（成功時 JSON例）：

```
{
  "recordId": 101,
  "userId": 5,
  "workDate": "2025-10-30",
  "inTime": "2025-10-30T09:00:00",
  "outTime": null,
  "status": "PunchedIn"
}
```

説明: 出勤打刻が成功すると、新規作成された勤怠レコードの情報を返します。`status` はアプリ側での状態管理用に含めています（PunchedIn: 出勤済、PunchedOut: 退勤済など）。既に出勤打刻済みの場合などはHTTP 400で以下のようなエラーを返します:

```
{ "error": "既に出勤打刻済みです。" }
```

• 休暇申請登録 API (**POST** /api/leave/requests):

リクエストボディ (JSON)：

```
{
  "leaveType": 1,
  "startDate": "2025-11-10",
  "endDate": "2025-11-12",
  "reason": "私用のため"
}
```

レスポンス（成功時 JSON例）：

```
{
  "requestId": 55,
  "status": "Pending",
  "message": "休暇申請を受け付けました。"
}
```

説明: リクエストには休暇の種類や期間等を指定し、成功時には新規発行された申請IDとステータスを返します。status は"Pending"（承認待ち）となります。バリデーションエラー時にはHTTP 400でエラーメッセージ:

```
{ "error": "開始日は必須です。" }
```

のように返します。

• 休暇承認 API (PUT /api/leave/requests/{id}/approve):

リクエストボディ: 例としてコメントを付与できる仕様の場合:

```
{ "comment": "プロジェクト引継ぎを確認しました。" }
```

レスポンス（成功時 JSON例）：

```
{
  "requestId": 55,
  "status": "Approved",
  "message": "休暇申請を承認しました。"
}
```

説明: 管理者が休暇申請ID 55を承認した場合の例です。適切な権限があり未承認の申請であればステータスが更新されます。既に処理済みの申請に対して承認APIを呼ぶとHTTP 409 Conflict等でエラーを返します。

• 勤怠月次レポートCSV API (GET /api/reports/attendance/monthly):

リクエストパラメータ: ?year=2025&month=10&deptId=3 など

レスポンス: text/csv のファイル (HTTPヘッダ Content-Disposition: attachment; filename="Attendance_202510_dept3.csv")

説明: このAPIはCSVファイルを直接応答します。取得したCSVの内容は例えば以下のようになります (レスポンスボディ例) :

```
日付,氏名,出勤時刻,退勤時刻,総労働時間,休暇種別
2025/10/01,田中太郎,09:00,18:00,8.0,
2025/10/02,田中太郎,09:15,18:00,7.75,遅刻
...
```

エラー時はHTTP 400または500を返し、ボディにはエラー説明のテキストを含めます。

ステータスコードとエラーレスポンス設計

APIではHTTPステータスコードを適切に利用し、クライアントが結果を判定しやすいようにします。主な使用コードと意味は以下の通りです。

- **200 OK:** GETやPUT等のリクエストが正常に処理された場合に使用。レスポンスボディには要求されたリソースまたは結果を含みます。
- **201 Created:** POSTで新規リソース作成が成功した場合に使用。レスポンスには作成されたリソースのIDやURLを含めます（ログインAPI等は除く）。
- **400 Bad Request:** クライアントからのリクエスト内容が不正（バリデーションエラー含む）の場合に使用。レスポンスボディには `{"error": "具体的なエラー内容"}` の形式でメッセージを返します。複数エラーがある場合はエラーリストを含めることも検討します。
- **401 Unauthorized:** 認証が必要なAPIに未認証でアクセスした場合、またはログイン失敗時に使用します。ボディには認証が必要な旨または認証失敗のメッセージを含めます。
- **403 Forbidden:** 認可されていない操作（例: 一般ユーザーが管理者APIにアクセス）に対して返します。メッセージ例: `{"error": "アクセス権がありません"}`。
- **404 Not Found:** 指定されたリソースIDに該当がない場合に返します。例えば存在しない休暇申請IDを操作しようとした場合など。
- **409 Conflict:** 業務的に矛盾した操作の場合に使用します。例えば二重打刻や、既に処理済みの申請を再度承認しようとした場合など、現在の状態とコンフリクトするリクエストに対して返します。
- **500 Internal Server Error:** サーバ側で予期せぬエラーが発生した場合に使用します。エラーレスポンスは `{"error": "サーバ内部でエラーが発生しました"}` のような一般メッセージとします（詳細はログにのみ記録）。

エラー時のレスポンスは基本的にJSON形式で、`error` プロパティにエラーの概要メッセージを入れる統一フォーマットとします（CSVダウンロードAPIなど一部例外を除く）。この統一により、フロントエンド側でエラーメッセージ処理を共通化できます。また、必要に応じて `errorCode` などを受け、国際化対応メッセージや詳細分類ができるよう拡張も考慮します。

まとめ

API設計では、社内Webシステムとしての利用だけでなく将来的なモバイル対応や外部連携を見据えてRESTfulなインタフェースを設けました。エンドポイントごとの機能と入出力フォーマットを定義し、HTTPステータスやエラーレスポンスの設計方針も示しました。これにより、クライアント側の実装やシステム連携時にも明確な取り決めのもとで通信が行えるようになります。堅牢なAPI設計はシステムの拡張性可他プラットフォーム対応力を高める基盤となります。

第8章：セキュリティ設計

認証・認可方式

本システムでは、社内システムという性質上、まずは**フォーム認証 + Cookie**を用いたセッション管理を基本とします。ユーザーがログイン画面で認証に成功すると、サーバ側でセッションを発行し、HttpOnly属性付きのセッションID Cookieをクライアントに付与します。以降のリクエストではこのCookieにより認証状態を維持します。

認可（アクセス制御）についてはASP.NET Coreの認可属性（`[Authorize]`属性）とロールベース認可を用います。具体的には、管理者用のコントローラ/アクションには `[Authorize(Roles="Admin")]` 属性を付与し、一般従業員にはアクセスできないようにします。逆に一般機能には `[Authorize]` のみを付与し、ログインさえしていればアクセス可能とします。

また、将来的なモバイルアプリや外部システム連携に備えて、**JWT(JSON Web Token)認証**もサポートします。ログインAPIでJWTを発行し、以降のAPIアクセスはHTTPヘッダにトークンを付与する方式です。JWTに

はユーザーIDやロールをクレームとして含め、APIサーバ側でトークンの署名と有効期限を検証します。これにより、セッションレスでスケーラブルな認証も可能となります。ただしJWT運用時はトークンの保護（HTTPS必須、長寿命トークンの失効管理など）に注意します。

パスワードのハッシュ化方針

ユーザーのパスワードは決して平文のまま保存しません。システムでは**ハッシュ化+ソルト**を用いてパスワードを保管します。具体的にはASP.NET Identityのデフォルトアルゴリズム（現在はPBKDF2によるハッシュ化）を利用するか、独自にSHA-256あるいはbcryptアルゴリズムでソルト付きハッシュを生成します。

新規登録やパスワード変更時には、パスワードの平文を受け取ったらただちにソルトを付与してストレッチング（反復計算）を行い、得られたハッシュ値（および必要ならソルト）をUsersテーブルのPasswordHashカラムに保存します。ログイン時には、入力されたパスワードに同じ処理を適用してハッシュを算出し、保存されたハッシュと比較します。

この方式により、万一データベースが漏洩した場合でもパスワードの逆算は非常に困難になります。さらにパスワードには大文字・小文字・数字・記号を混ぜた強度の高いものをユーザーに設定してもらうよう、UI上でポリシーを示します。初期パスワード発行時もランダムで推測されにくい値を用い、初回ログイン時の変更を促すなどセキュリティに配慮します。

入力値検証・脆弱性対策

- **入力値検証:** すべての外部からの入力に対してサーバ側で検証を行います。特にSQLインジェクション対策として、データアクセスにはパラメータバインド（Entity Framework Coreでは自動的にパラメータ化されます）を徹底し、動的にSQL文字列を組み立てることは避けます。OSコマンドインジェクションなどは本システムでは該当する機能はありませんが、万一ファイル操作等を行う場合も入力パスをサニタイズします。
- **XSS対策:** クロスサイトスクリプティングについては、ユーザー入力をそのまま画面に表示しない原則を守ります。ASP.NET CoreのRazorビューでは出力時に自動でHTMLエンコードが行われるため基本的なXSSは防げますが、例えば管理者が入力したスクリプトが従業員画面に表示されるようなケースでも問題が起きないように、すべての表示データをエスケープします。特に休暇申請の理由コメント等、ユーザーが自由記入するフィールドでは<や>等の特殊文字をエスケープして保存します。
- **CSRF対策:** クロスサイトリクエストフォージェリ対策として、ログイン後の重要な操作（休暇申請送信、打刻処理、ユーザー更新など）にはCSRFトークンを利用します。ASP.NET Core MVCではフォーム用ヘルパーにより自動的に隠しフィールドに抗CSRFトークンを埋め込み、サーバで検証する仕組みを導入します。これにより、悪意の第三者サイトからユーザーのセッションを悪用したリクエストを防ぎます。
- **その他のセキュリティヘッダ:** Webアプリとして、HTTPレスポンスヘッダに適切なセキュリティ指針を設定します。例として、Content-Security-Policy (CSP)ヘッダを設定し、信頼できないスクリプトやリソースの読み込みを制限します。また、X-Frame-Optionsヘッダでクリックジャッキングを防止し、X-Content-Type-OptionsでMIMEタイプの強制を有効にします。
- **セッション管理:** セッションIDには推測困難な乱数を使用し、CookieにはHttpOnly属性とSecure属性を付与してJavaScriptからのアクセスや平文通信での漏洩を防ぎます。一定期間操作がなければタイムアウトし、再ログインを要求します。
- **アクセス制御:** 機能レベルのアクセス制御として、サーバサイドでユーザーのロール・権限を都度確認します。例えばAPIレベルでも、休暇承認APIは管理者ロールを持つJWTでなければHTTP 403を返すようにします。UI上非表示でも、バックエンド側でチェックを二重に行うことで不正アクセスを排除します。
- **ログイン試行制限:** 前述した通り、一定回数（例えば5回）連続でログイン失敗したユーザーアカウントは自動ロックします。Usersテーブルにロックアウトフラグやロック解除予定時刻を設け、例えば15分間ロックする運用とします。これにより総当たり攻撃を緩和します。なお管理者にはロック解除操作も提供します。

まとめ

セキュリティ設計では、認証・認可からデータ保護、脆弱性対策まで多層的に配慮しました。Cookieベースのセッション管理とJWTの組み合わせにより利便性と拡張性を両立し、パスワードの安全な取り扱いや入力データの検証で基本的な攻撃ベクトルを遮断します。また、Web標準のセキュリティ対策（XSS, CSRF対策など）を盛り込み、ログイン試行制限など実運用でのリスクにも対応しました。これらの実装により、ユーザーの大切な勤怠データを守り、安全にシステムを運用できるようになります。

第9章：ログ・監視設計

ログ出力方針

システムの動作を追跡し、問題発生時に迅速に原因分析するため、適切なログ出力を行います。本システムではログレベルを以下のように運用します。

- **DEBUG:** 開発時や詳細分析用の情報を出力します。本番環境では通常出力しません。例: メソッドの入り口・出口、変数の値等。
- **INFO:** 通常の操作記録やシステムの状態を出力します。ユーザーの主なアクション（ログイン成功、打刻処理成功、休暇申請登録など）はINFOレベルで記録します。例: 「User 5 punched in at 09:00」。
- **WARN:** 想定外だが重大ではない事象を記録します。処理自体は続行可能だが注意が必要な場合（例: 一時的な外部サービスタイムアウトからのリトライ成功、ユーザー入力がおかしかったがデフォルト値にフォールバックした等）。
- **ERROR:** 処理不能なエラーや重大な問題を記録します（例: DB接続失敗、未処理例外キャッチ時）。スタックトレースやセッション情報など、問題解決に必要なデータを可能な範囲で残します。
- **FATAL:** システム全体の停止を招く致命的なエラー時に使用します（想定としてはほぼありませんが、例えば構成ファイルが読み込めない等でアプリが起動できない場合）。

ログの出力先は、テキストファイルおよびコンソール（Docker運用時など）の両方に行います。ログフォーマットは日付・時刻、レベル、イベント内容、関連ID等を含むよう統一します。例えばシンプルなテキストログでは:

```
2025-10-30 09:00:00 [INFO] UserId=5 Action=PunchIn WorkDate=2025-10-30 Status=Success
```

のように出力します。さらに分析容易性のため**構造化ログ**も検討します。JSON形式でログを吐き出す設定を行えば、例えば:

```
{
  "timestamp": "2025-10-30T09:00:00",
  "level": "INFO",
  "event": "PunchIn",
  "userId": 5,
  "workDate": "2025-10-30",
  "message": "Punch in successful"
}
```

のようなログを蓄積でき、ログ監視ツールでの検索や集計が容易になります。

アクセスログ（HTTPリクエストログ）についても必要に応じて記録します。ただし社内システムで流量も限

られるため、IIS等の標準アクセスログに任せ、アプリケーションでは重要操作のイベントログを中心に残します。

監視項目

システム稼働中に監視すべき重要な指標・イベントを定義します。小規模システムとはいえ、予兆を捉えることで大きな障害を防ぐことができます。

- **エラー監視:** エラーログ（ERROR以上のログイベント）の発生を監視します。例えば1時間に一定回数以上ERRORが出現した場合にアラートを上げるなどの設定を行います。特にDB接続エラーやNullReferenceException等、重大なものは即時通知レベルとします。
- **パフォーマンス監視:** WebサーバおよびデータベースサーバのCPU使用率、メモリ使用量、ディスクIOなどを定期監視します。レスポンスタイムの監視も実施し、主要APIやページの応答時間が通常より長くなっていないかチェックします（APMを導入してスロークエリや遅延APIのトレースも検討します）。
- **稼働監視（死活監視）:** サーバプロセスが稼働しているか、定期的なHTTPヘルスチェックエンドポイント（例: `/health`）へのリクエストで確認します。応答がなければシステム管理者に通知します。
- **ログイン/セキュリティ監視:** 異常なログイン試行（短時間に多数の失敗など）を検知します。例えば単一アカウントに対する連続失敗や、多数のアカウントへの辞書攻撃的試行をログ解析で探知し、管理者に警告します。
- **容量監視:** ログファイルやデータベースのディスク使用量を定期チェックします。特にCSVレポート出力機能があるため、一時ファイルや出力先ディレクトリが肥大化していないかも確認します（適宜古いファイルのクリーンアップを実施）。
- **バッチ/タスク監視:** 将来的にバッチ処理や定期タスク（例えば日次でメール通知を送る等）が導入された場合、その成否や実行時間をログに記録し、失敗時には通知します。

これらの監視は、Azure Monitorやクラウド監視サービス、またはZabbix等のオンプレ監視ツールを用いて実装します。閾値や検知内容は運用開始後にチューニングし、誤検知を減らしつつ迅速な問題検知を目指します。

ログ出力例

以下に、実際のログ出力の例を示します。構造化ログを想定したJSON形式の例と、プレーンテキストの例を記載します。

構造化ログの例（JSON形式）:

```
{
  "timestamp": "2025-11-01T08:59:59",
  "level": "INFO",
  "userId": 12,
  "userName": "tanaka",
  "event": "LoginSuccess",
  "message": "User tanaka (ID=12) logged in successfully from IP 192.168.1.10"
}
```

```
{
  "timestamp": "2025-11-01T09:00:05",
  "level": "ERROR",

```

```
"userId": 12,  
"event": "PunchOut",  
"message": "Failed to punch out: no open attendance record",  
"errorDetail": "AttendanceNotFoundException: No record found for userId=12 date=2025-11-01"  
}
```

テキストログの例:

```
2025-11-01 09:00:00 [INFO] UserID=5 Action=PunchIn Success (InTime=2025-11-01T09:00:00)  
2025-11-01 18:00:10 [INFO] UserID=5 Action=PunchOut Success (OutTime=2025-11-01T18:00:00,  
WorkHours=8.0)  
2025-11-01 18:01:00 [WARN] UserID=5 Action=Login Attempt failed (Invalid password) Count=1  
2025-11-01 18:05:30 [ERROR] Action=SQLExecute Query=GetMonthlySummary  
Message=TimeoutException (Duration=31s)
```

上記例では、INFOレベルで正常な打刻やログインを、WARNで怪しいログイン試行、ERRORでSQLタイムアウトエラーを記録しています。エラーには例外の種類や処理時間など詳細を含め、後続の分析に役立つようにしています。

まとめ

ログと監視の設計によって、システム運用中の可観測性を高めています。適切な粒度でのログ出力はトラブルシューティングを迅速にし、監視項目の設定は障害の予防と早期発見につながります。本システムでは、業務イベントのログやエラー検知を通じて、信頼性の高い運用を支援します。将来的に規模拡大した場合でも、ログ/監視基盤を拡充することでスムーズに対応できる土台を構築しています。

第10章：テスト設計

テスト戦略

本システムの品質を確保するため、段階的にテストを実施します。主なテストレベルと戦略は以下の通りです。

- **単体テスト（ユニットテスト）**：ビジネスロジック層やユーティリティ関数など、個々のモジュールに対してテストを行います。例えば、勤怠の集計計算関数、休暇申請のバリデーションロジック、残業時間計算ロジックなどを対象にします。フレームワークとしてはxUnitやMSTestを用い、テスト駆動開発（TDD）的にロジックを実装します。依存するリポジトリなどはモック（Moq等利用）し、純粹にロジックの正しさを検証します。
- **結合テスト（インテグレーションテスト）**：システムの各層が連携して正常に動作するかを確認します。例えば、実際にEntity Framework Coreでインメモリのデータベース（またはローカルのテスト用DB）に接続し、リポジトリ経由でデータ登録・取得が期待通り動くか、コントローラをテストホスト上で起動しHTTPリクエストからレスポンスまで一連の流れをテストする、といった内容です。Webアプリのルーティングやフィルタ、認証処理も含め検証します。
- **システムテスト（総合テスト）**：テスト環境上にデプロイしたシステムを用いて、実際のユーザー操作をシナリオに沿って検証します。QAエンジニアや担当者がブラウザ上で画面遷移や機能を手動テストするほか、Selenium等を用いた自動UIテストも検討します。例えば「従業員が出勤打刻し、管理者がその日の勤怠を確認できる」一連の流れや、「休暇申請～承認」フローのテストケースを実行します。

- ・**負荷テスト（パフォーマンステスト）**：小中規模向けシステムですが、ピーク時に問題なく動作するかを確認します。想定同時ユーザー数（例: 50人が9:00に一斉に打刻）での動作をJMeter等でシミュレーションし、レスポンスが許容範囲内か、サーバリソースに余裕があるかを計測します。ボトルネックが分かればチューニングを検討します。

テスト項目例

各主要機能に対して、代表的なテストケースをいくつか挙げます。

- ・**ログイン機能:**
 - ・正しいユーザーIDとパスワードを入力した場合にログイン成功し、想定ページにリダイレクトされる。
 - ・誤ったパスワードの場合にエラーメッセージが表示され、ログインできない。
 - ・存在しないユーザーIDの場合にエラーメッセージが表示される。
 - ・ログイン試行失敗5回でアカウントがロックされ、以降正しいパスワードでもログイン不可となる（ロックアウトの確認）。
- ・**出退勤打刻機能:**
 - ・未出勤の状態で「出勤」打刻すると、勤怠レコードが作成されステータスが「出勤済み」になる。
 - ・既に出勤済みの状態で再度「出勤」を押すとエラーメッセージが表示され処理されない（二重打刻防止）。
 - ・出勤後、同じ日に「退勤」打刻すると対応するレコードの退勤時刻が記録され勤務時間が計算される。
 - ・出勤していない日に「退勤」を押した場合、エラーとなり打刻できない。
- ・**日次勤務一覧/月次集計:**
 - ・勤怠データがある期間を表示したとき、各日付の出退勤が正しく表示され合計も正しい。
 - ・勤怠データが無い場合「記録なし」のメッセージが表示される。
 - ・管理者が部下のデータを閲覧するとき、自分の部下以外のユーザーは選択肢に出ず不正閲覧できない（認可確認）。
- ・**休暇申請/承認:**
 - ・必須項目を全て正しく入力して申請するとデータベースに申請が登録され、申請者のステータスが「未承認」で取得できる。
 - ・開始日>終了日のような不正期間の場合、申請が受け付けられず画面にエラーが表示される。
 - ・管理者が承認画面で「承認」すると該当申請のステータスが「承認済」となり、申請者が確認したときに反映されている。
 - ・既に否認済みの申請に再度承認APIを呼んでも処理されない（結果が変更されない）。
- ・**ユーザー管理:**
 - ・新規ユーザー登録画面で必要情報を入力し保存すると、Usersテーブルに新レコードが作成され一覧に表示される。
 - ・重複するメールアドレスで登録しようするとエラーメッセージが表示され登録できない。
 - ・ユーザー編集でロールを一般→管理者に変更すると、そのユーザーが次回ログイン時に管理者メニューへアクセスできるようになる（ロール反映の確認）。
 - ・ユーザー無効化を行うと、そのユーザーはログインできなくなる（無効フラグの確認）。
- ・**セキュリティ関連:**
 - ・認証が必要なページに未ログインでアクセスするとログイン画面にリダイレクトされる。
 - ・一般ユーザーで管理者専用URLにアクセスすると403拒否される。
 - ・CSRFトークンを改ざん・不正化してフォーム送信した場合、サーバ側でリクエストが拒否される。
 - ・XSS攻撃を狙ったスクリプトタグ入りの入力（例えば休暇理由に `<script>alert('xss')</script>` を入れる等）をしても、画面表示時にスクリプトが実行されない（エスケープされて表示される）。

テストデータ例

テストを実施するにあたり、想定されるデータの組み合わせを準備します。以下に一部例を示します。

- ・**ユーザーテストデータ:**

- ・`user_admin` (ID=1): 管理者ロール、氏名「管理 太郎」、メール `admin@example.com`、部下に一般ユーザーを持つ。
- ・`user_emp1` (ID=2): 一般従業員ロール、氏名「社員 一郎」、メール `ichiro@example.com`、上長 `user_admin`。
- ・`user_emp2` (ID=3): 一般従業員ロール、氏名「社員 二郎」、メール `jiro@example.com`、上長 `user_admin` (複数部下シナリオ)。
- ・退職済ユーザー `user_retired` (ID=4): `isActive=false`、ログイン不可データ。

- ・**勤怠レコードテストデータ:**

- ・`user_emp1`について2025/10/01～2025/10/05の5日間連続勤務データ (9:00-18:00、休憩1h想定で実働8h) を登録。10/03のみ残業1時間 (退勤19:00) など変化をつける。
- ・`user_emp2`については勤務なし日 (欠勤) や遅刻 (10:00出勤) データを含め、月次集計にばらつきが出るようにする。

- ・**休暇申請テストデータ:**

- ・`user_emp1`が2025/11/10～2025/11/12の3日間有給休暇を申請したレコード (Status=0未承認) 。
- ・`user_emp2`が単日(2025/11/15)特別休暇申請したレコード (Status=1承認済) —テスト開始前に承認処理も実施。
- ・過去日付(2025/01/05)の休暇申請レコード (理論上有ってもよいが承認済とかで配置) 。

- ・**その他:**

- ・ログイン試行履歴用に、`user_emp1`で4回パスワードミスした直前の状態などをセット。
- ・ユーザー管理用にダミーユーザーを10件程度事前登録し、検索・ページングの挙動を確認。

上記テストデータを用いることで、多様なケースに対するシナリオを網羅します。テストケースごとに期待結果 (Expected Result) を定義し、実行結果と比較します。不具合が見つかった場合は原因を分析し、設計や実装を見直します。また、テストは単発で終わらず、継続的インテグレーション (CI) 環境で自動実行することで、将来の変更によるリグレッションを早期発見します。

まとめ

テスト設計では、システムが期待通りに動作し、また不正な操作や負荷状況でも安定していることを検証する計画を示しました。単体から結合、システムテストまで多層的に行うことで不具合混入を防ぎます。代表的なテスト項目とデータ例を挙げることで、網羅性の高いテストケースを準備しています。十分なテストを経てリリースすることで、信頼性の高い勤怠管理システムをユーザーに提供できるようにします。

第11章：保守・拡張性設計

コードのモジュール化・再利用性設計

本システムでは、前述の通りレイヤードアーキテクチャを採用し関心事の分離を図っています。そのため、コードも機能ごと・責務ごとにモジュール化されています。サービスクラスやリポジトリクラスは単一責任の原則に従い、小さくまとまっているため再利用が容易です。例えば、勤務時間計算ロジックは

`WorkTimeCalculator` のようなクラス/メソッドに抽出し、出退勤処理や月次集計処理など複数箇所から呼び出せるようにします。

また、**依存性の注入 (DI)** を活用して、モジュール間の結合度を下げています。インターフェースを定義し、実装を差し替えられるようにすることで、将来的にデータベースを変更したり外部APIを導入したりする際にも、インターフェース実装を用意するだけで他のコードに影響を及ぼさない設計としています。これによりユニットテスト時にモック実装を注入してテストすることも容易になり、保守性が高まります。

UI部分でも、共通コンポーネント（ヘッダーやメニュー、入力フォーム部品など）を部分ビューやタグヘルパーとして切り出し、重複コードを減らしています。これによりデザイン変更時には一箇所修正すれば全画面に反映されるため、保守コストを削減できます。

総じて、機能追加の際には既存コードを流用・拡張しやすい構造となっており、一から書き直す必要がないよう配慮しています。例えば、新たに「勤怠承認」機能（上長が部下の勤怠を承認するワークフロー）を追加する際も、既存の休暇承認機構を参考にロジックを共通化する、といった進め方ができるでしょう。

設定値の外部化

システム中に埋め込まれる様々な定数や設定値は、可能な限りコードから分離し外部ファイルや環境変数で管理します。ASP.NET Coreでは `appsettings.json` および環境ごとの設定ファイル（`appsettings.Development.json` 等）を用いるため、これを積極的に活用します。

具体的には、以下のような項目を外部設定とします。 - **データベース接続文字列:** データベースサーバ名、認証情報などを `appsettings.json` に記載（本番環境では環境変数やキーボルトークンで上書き）。 - **アプリケーション設定:** 例: ログイン試行制限の回数やロックアウト時間、セッションタイムアウト時間、1日の標準労働時間(8時間)など、今後変更の可能性があるビジネスルール値。 - **パス/URL:** CSV出力先のパス（またはストレージ接続情報）、外部サービスのエンドポイント（メールSMTPサーバ等）など、環境により異なる値。 - **機能トグル:** 新機能を段階的に有効化/無効化するフラグを設定ファイルで管理し、運用中に切り替えられるようにする（Feature Toggleの考え方）。

設定ファイルはインフラ担当者や開発者が編集でき、アプリケーション再起動なく適用可能なものはHot Reloadや設定リロード機構を使います。また、センシティブな情報（DBパスワード等）はソース管理に載せないようにし、環境変数やAzure Key Vaultなどから読み込むようにします。

これらの工夫により、コードを変更せずに設定値を調整できるため、環境移行や要件変更に対応可能です。例えば会社の就業規則で標準労働時間が変わった場合も、設定値を書き換えるだけでシステム計算結果を更新できます。

今後の拡張ポイント・スケーラビリティ設計

本システムは現時点では小中規模向けの想定ですが、将来的な拡張や利用者増にも耐えられるよう設計段階でいくつかの拡張ポイントを考慮しています。

- ・**機能拡張:** 勤怠管理機能に関連して、将来的にシフト管理機能、残業申請・承認機能、プロジェクト別工数管理との連携などが考えられます。現在の設計ではドメインオブジェクトやサービスを追加するだけで容易に新機能を組み込めます。例えばシフト管理ではShiftエンティティとスケジュール生成ロジックを追加し、既存AttendanceRecordsと組み合わせると勤務実績との比較ができます。休暇承認のワークフローを参考に残業申請承認を実装することも可能です。
- ・**ユーザー数増加:** 同時ユーザー数が大幅に増えた場合、スケールアウトやチューニングで対応できます。ASP.NET Coreアプリはステートレスに設計（セッションも必要最小限に）されているため、Webサーバを複数台構成しロードバランサ配下に置くことで高負荷に対応できます。セッション管理を分散環境向けにRedis等に移行することも容易です。データベースもAzure SQLへの移行や、読み取り負荷が高ければリードレプリカの導入などでスケールします。
- ・**クラウド対応:** 初期はオンプレミス想定でも、AzureやAWS等へのデプロイを視野に入れています。コンテナ化（Docker）も可能なため、Kubernetes上での運用も検討できます。クラウドサービス（Azure ADでのSSO認証、Azure Functionsでのバッチ処理等）との統合も、現在のアーキテクチャを維持しつつ追加できます。
- ・**多言語対応:** 今後、外国人従業員の増加等によりシステムの多言語化が必要になった際も、ASP.NET Coreのローカリゼーション機構(Resourceファイル等)を用いてUI表示を切り替え可能です。画面設計では文字列を直書きせずリソース参照することで対応しやすくしてあります。
- ・**他システム連携:** 人事システムや給与計算ソフトとの連携要求が出た場合、現在のAPI層を拡充し、必要なデータを提供するWeb APIエンドポイントを増やすことで実現できます。また、CSVインポート/

エクスポート機能を追加して、他システムとのデータ受け渡しも可能です。設計段階でデータ構造を業務標準（例えば日本の勤怠CSVフォーマット）に沿うよう考慮しておくことで、連携コストを下げています。

- **セキュリティ強化:** 社内システムとはいえ、将来的に在宅勤務増加等で外部からアクセスさせる場合はVPNやWAFの導入も検討されます。本システム自体も2要素認証の導入（例えばワンタイムパスコード送信）やより厳格な監査ログ（ユーザー操作履歴の詳細記録）などを追加できるよう拡張性を確保しています。
- **モジュール分割:** 機能が大規模化した際は、マイクロサービス化やBFF(Backend for Frontend)の導入でモジュールごとにサービスを分けることも視野に入れます。例えば勤怠打刻サービスと休暇管理サービスを別にし、それぞれスケールや開発を独立させることも可能です。現在のコード構造は明確に境界が分かれているため、分離もしやすくなっています。

このように、本設計では将来を見据え、必要に応じてスムーズに拡張・スケールできる柔軟性を持たせています。現在必要ない複雑さは実装していませんが、いざという時に備えてアーキテクチャ上の余裕を確保しておくことで、システムのライフサイクル全体での長期的な有用性を保証します。

まとめ

保守と拡張性の設計では、現在の要件を満たすだけでなく今後の変化にも対応できるような工夫を盛り込みました。コードのモジュール化により変更影響を局所化し、設定値の外部化で動的な調整を可能にしています。また、システム規模や要件の拡張にも柔軟に対応できるよう、アーキテクチャや技術選定の段階で将来性を意識しました。これらにより、リリース後の保守作業を効率化しつつ、新たなニーズにも迅速に応えられる基盤を構築しています。

第12章：付録

用語集

本システム設計書内で使用した専門用語や略語について解説します。

- **勤怠（きんたい）:** 出勤と退勤など、従業員の勤務状況全般を指す言葉です。「勤怠管理」は勤務時間や休暇の管理業務のこと。
- **打刻（だこく）:** 出勤時刻・退勤時刻を記録すること。タイムカードを押すイメージから来ています。システム上ではボタン操作等で時間を記録する行為を指します。
- **MVC (Model-View-Controller):** ソフトウェアアーキテクチャパターンの一つで、本システムのWebアプリケーションフレームワークとして採用しています。Model（モデル）、View（ビュー）、Controller（コントローラ）に機能を分けることで開発と保守を効率化します。
- **ASP.NET Core MVC:** Microsoftが提供するWebアプリケーションフレームワーク。C#言語でサーバサイドロジックを書き、MVCパターンでWebシステムを構築できます。
- **Entity Framework (Core):** .NET向けのO/Rマッパー（Object-Relational Mapper）です。データベースのレコードをオブジェクトとして扱えるようにし、SQLを意識せずにデータアクセス処理を記述できます。
- **REST API:** Web上でリソースを操作するための設計様式に則ったAPIのことです。HTTPメソッド（GET, POST, PUT, DELETEなど）を用いてCRUD操作を行います。本システムでも一部機能でRESTfulなAPIを提供します。
- **JWT (JSON Web Token):** 認証に使われるトークンの一種で、JSON形式のクレーム（ユーザー情報など）を含む文字列です。電子署名により改ざん検知が可能。認証済みユーザーの状態をクライアント側に保持する際に使われます。
- **CSRF (Cross-Site Request Forgery):** クロスサイトリクエストフォージェリ攻撃の略称。ユーザーが意図しないリクエストを別サイト経由で実行させられる攻撃手法。本システムではトークン検証により対策。

- **XSS (Cross-Site Scripting):** クロスサイトスクリプティング攻撃。悪意あるスクリプトをWebページ上で実行させ、情報漏洩等を狙う攻撃手法。本システムでは出力エスケープ等で対策。
- **CI/CD:** 継続的インテグレーション/継続的デリバリの略。コードの自動ビルド・テスト・デブロイのパイプラインを構築する手法です。本プロジェクトではGitHub Actions等でCIを実施予定。
- **ER図 (Entity-Relationship 図):** データベース設計において、エンティティ（テーブル）間の関係を図式化したもの。本章ではテキストでそれを表現しました。
- **リポジトリ (Repository):** ソフトウェアパターンの一つで、データアクセス用のクラス/インターフェースを指します。データストアへの問い合わせや保存ロジックをカプセル化します。
- **デブロイ:** 開発したシステムを実際のサーバや環境に配置し、利用可能な状態にする作業。リリースとも言います。
- **負荷テスト:** システムに想定以上のアクセスや処理をかけて、性能や安定性を確認するテストです。
- **ローカリゼーション (i18n/l10n):** アプリケーションを多言語対応させること。i18nはInternationalizationの略で国際化、l10nはLocalizationの略で現地化とも言います。

参照ドキュメント・参考資料

システム開発にあたり、以下のドキュメントや資料を参照しています。

- **勤怠管理システム 要件定義書 (社内ドキュメント)** – 本システムに求められる業務要件をまとめた文書。機能一覧や業務フロー、画面項目定義などが記載されています。
- **労働基準法 関連資料** – 勤怠管理に関わる法定項目（労働時間管理、有給休暇管理の義務など）を確認するため、厚生労働省のガイドラインや関連Webサイトを参照しました。
- **ASP.NET Core ドキュメント (Microsoft Docs)** – ASP.NET Core MVCや認証、Entity Framework Coreの公式ドキュメント。設計上のベストプラクティスや実装方法の確認に使用。
- **開発標準ガイドライン (社内)** – コーディング規約やアーキテクチャ指針を示した社内ドキュメント。本プロジェクトでもこれに準拠して設計・実装を行います。
- **UIデザインガイド** – ユーザビリティとデザイン統一のために参照した資料。特にBootstrap等のコンポーネントの使い方や、レスポンシブデザインのポイントを確認しました。

リポジトリ構成

本プロジェクトのソースコードおよび関連ファイルはGitによってバージョン管理されています。リポジトリ内の構成は以下のようになっています。

```
project-root/ (リポジトリのルートディレクトリ)
├── src/                ソースコードおよびソリューションファイル
│   ├── AttendanceSystem.sln      ソリューションファイル
│   ├── AttendanceSystem.Web/      Webアプリ (ASP.NET Core MVCプロジェクト)
│   ├── AttendanceSystem.Core/     コアロジック (クラスライブラリプロジェクト)
│   ├── AttendanceSystem.Infrastructure/ インフラ層 (クラスライブラリプロジェクト)
│   └── AttendanceSystem.Tests/     テストプロジェクト (単体・結合テスト)
├── docs/              プロジェクト関連ドキュメント
│   ├── requirements_spec.docx     要件定義書
│   ├── system_design.md          本設計書 (Markdown版)
│   └── test_plan.xlsx            テスト計画・項目一覧
├── scripts/           デブロイやDBマイグレーション用スクリプト
├── .github/workflows/ CI設定 (GitHub Actions のワークフローファイル)
└── README.md          リポジトリREADME (セットアップ方法やプロジェクト概要)
```

ブランチ運用は `main`（または `master`）が安定版、`develop` が次リリース準備中のブランチとして運用されています。機能ごとにブランチを切り、コードレビュー後に `develop` へ統合、最終リリース時に `main` へマージするフローです。

タグはバージョン番号に沿って付与し、`v1.0`、`v1.1` のようにリリースごとのソースを識別可能としています。また、Pull Requestベースでの開発を行い、CIによってテストが自動実行される仕組みが設定されています。

まとめ

付録では、本文中で使用した用語の説明や、参照した情報源、そしてプロジェクトの具体的な構成について整理しました。これにより、本設計書の内容を読み解く手助けとなり、開発チーム内外の関係者との共通理解を促します。リポジトリ構成の提示によって、実際の開発物の所在も明確になり、ドキュメントと実装を行き来しやすくなっています。本資料が、本システムの開発と保守運用における道しるべとなることを期待します。
