Projeto de conclusão de curso Itera360

Leonardo Wagner Oliveira Dias

*Portal do Funcionário*

Alfenas-MG

2024

Leonardo Wagner Oliveira Dias

Projeto

*Portal do funcionário*

A conexão rápida com sua empresa

Trabalho apresentado no curso

profissionalizante da Itera360.

Alfenas-MG

2024

**Introdução**  
  
O projeto desenvolvido consistiu na criação de um **portal de gerenciamento para uso dos funcionários**, com o objetivo de facilitar o acesso a informações pessoais e administrativas por parte dos colaboradores de uma empresa. A aplicação permite que os usuários visualizem e atualizem seus dados pessoais, agendem férias, consultem o banco de horas e suas horas extras, verifiquem e tenham acesso aos holerites.

Para isso, a solução foi construída utilizando **React** para o desenvolvimento do frontend, proporcionando uma interface dinâmica e responsiva. No backend, foi utilizado **.NET Core**, garantindo robustez e escalabilidade na gestão dos dados. O sistema foi integrado a um banco de dados **SQL Server**, permitindo armazenamento seguro e eficiente das informações.

A ideia surgiu para facilitar, reduzir custos e aumentar a eficiência na relação entre empresa e colaborador. Proporcionando assim, que o colaborador possa consultar informações administrativas ou até mesmo alterar dados cadastrais. Ações que dependeriam do intermédio da empresa, ele estaria resolvendo por si só, poupando tempo e trabalho de terceiros e dele próprio.

* **Objetivos do projeto**

***Facilitar o Acesso às Informações Pessoais:*** Prover aos funcionários uma plataforma centralizada onde possam visualizar e atualizar de forma autônoma seus dados pessoais, como nome, endereço, telefone e outros dados relevantes.

***Automatizar Processos de Gestão de Recursos Humanos:*** Automatizar tarefas comuns do departamento de RH, como o agendamento de férias, controle de banco de horas e registro de horas extras, reduzindo a carga de trabalho manual e minimizando erros.

***Proporcionar Segurança e Integridade dos Dados:*** Garantir que todas as informações armazenadas sejam protegidas por mecanismos de segurança robustos, assegurando a privacidade dos dados dos usuários e a conformidade com normas de proteção de dados.

***Melhorar a Comunicação entre Empresa e Funcionários:*** Facilitar a comunicação entre a empresa e seus colaboradores através de notificações e atualizações de dados, tornando o processo de gestão de funcionários mais transparente e eficiente.

***Promover a Usabilidade e Acessibilidade:*** Desenvolver uma interface de usuário intuitiva e responsiva, garantindo que o sistema seja fácil de usar, acessível em diferentes dispositivos e adaptável às necessidades dos diversos perfis de usuários.

***Permitir a Escalabilidade e Manutenção do Sistema:*** Construir a aplicação de forma modular e escalável, permitindo que futuras expansões e integrações sejam realizadas de forma ágil, sem comprometer a estabilidade do sistema existente.

* **Alcance do Projeto:**

***Funcionalidades Centrais***: O sistema será desenvolvido para permitir que os usuários gerenciem informações pessoais, agendem férias, visualizem o banco de horas, acessem holerites e acessem as horas extras registradas. Estas funcionalidades serão acessíveis através de uma interface intuitiva, desenvolvida em React.js, conectada ao back-end em .NET Core.

***Responsividade:* foi uma preocupação ao desenvolver o projeto que alcançasse o maior número de usuários possível. A ideia principal é que, apesar de poder ser usado em computadores de mesa, sua usabilidade em dispositivos móveis fosse otimizada, pois esses dispositivos são usados pela maioria do público alvo do projeto.**

***Gerenciamento de Usuários***: O sistema permitirá que os usuários atualizem seus dados pessoais, alterem sua senha e desativem suas contas, conforme necessário.

***Escalabilidade***: O sistema foi projetado para ser escalável, o escopo atual prevê a implementação de recursos como mais opções de usabilidade e disparo de avisos, por exemplo. Estes aspectos seriam abordados em fases futuras do projeto.

* **Limitações do Projeto:**

***Integrações Externas***: O projeto não inclui a integração com sistemas de terceiros, como sistemas de folha de pagamento externos. Todas as operações serão realizadas dentro do escopo do próprio sistema.

***Manutenção e Suporte***: O escopo não abrange um plano detalhado de manutenção e suporte pós-implantação, focando apenas na entrega do produto final.

**Visão geral do sistema**

* **Arquitetura do sistema**

A arquitetura do sistema foi cuidadosamente planejada para garantir escalabilidade, segurança e eficiência. O sistema é construído utilizando uma arquitetura de microservices, dividida em várias camadas, cada uma com responsabilidades bem definidas:

***Camada de Apresentação***: Esta camada é desenvolvida em React.js e é responsável pela interface do usuário, oferecendo uma experiência de uso intuitiva e interativa. Ela consome as APIs do back-end para exibir dados e executar ações conforme solicitado pelos usuários.

***Camada de Aplicação (Back-end*)**: Desenvolvida em .NET Core, esta camada atua como o núcleo do sistema, onde reside a lógica de negócios. Os serviços são expostos através de APIs RESTful, permitindo comunicação eficiente entre a interface do usuário e o banco de dados. Esta camada é responsável por processar solicitações, aplicar regras de negócio e retornar as respostas apropriadas.

***Camada de Persistência (Banco de Dados):*** O sistema utiliza o SQL Server como banco de dados relacional para armazenar informações críticas, como dados de usuários, registros de horários e detalhes de folhas de pagamento. A comunicação entre a camada de aplicação e o banco de dados é feita através de ORM (Object-Relational Mapping), utilizando Entity Framework Core.

***Autenticação e Segurança:*** A arquitetura inclui mecanismos de autenticação baseados em usuário e senha, para garantir que apenas usuários que tenham o CPF cadastros possam acessar determinadas funcionalidades. Além disso, todas as comunicações entre as camadas utilizam HTTPS para assegurar a integridade e confidencialidade dos dados.

* **Funcionalidades**

As funcionalidades atuais do sistema foram desenhadas para atender às principais necessidades dos usuários em um ambiente corporativo:

***Gerenciamento de Informações Pessoais:*** Os usuários podem atualizar seus dados pessoais, como nome, sobrenome, telefone e endereço. Essas informações são armazenadas com segurança no banco de dados e podem ser modificadas diretamente pelo usuário.

***Agendamento de Féria (em desenvolvimento):*** O sistema permite que os usuários planejem e agendem suas férias, com a possibilidade de verificar a disponibilidade e evitar conflitos de datas.

***Banco de Horas (em desenvolvimento):*** Os usuários podem visualizar e gerenciar seu banco de horas, acompanhando as horas trabalhadas além do expediente normal e compensações futuras.

***Acesso a Holerites:*** A funcionalidade de visualização de holerites permite que os usuários acessem suas folhas de pagamento de maneira segura, com um histórico acessível a qualquer momento.

***Registro de Horas Extras (em desenvolvimento):*** O sistema facilita o acesso às horas extras, permitindo que o usuário consulte quantidades e valores.

***Segurança e Gestão de Acesso:*** Os usuários podem alterar suas senhas de forma segura e, caso necessário, desativar suas contas diretamente pelo sistema.

Essas funcionalidades asseguram que o sistema atenda às necessidades operacionais e administrativas dos usuários, proporcionando uma plataforma integrada para o gerenciamento de informações pessoais e profissionais.

**Configurações do ambiente**

* **Requisitos de software e hardware**

**Requisitos de Software:**

***Sistema Operacional:*** Windows 10 ou superior, macOS 10.15 ou superior, ou uma distribuição Linux recente.

***.NET Core SDK***: Versão 6.0 ou superior.

***Node.js***: Versão 14.x ou superior.

***NPM (Node Package Manager):*** Incluído com a instalação do Node.js.

***React.js***: Utilizado no front-end (não requer instalação separada, gerenciado via npm).

***SQL Server:*** Versão 2017 ou superior, para o ambiente de produção e desenvolvimento.

***Visual Studio Code ou Visual Studio 2019/2022***: Para edição de código.

***Git***: Para controle de versão.

**Requisitos de Hardware:**

***Processador***: Intel i5 ou equivalente, 2.5 GHz ou superior.

***Memória RAM***: Mínimo de 4 GB (16 GB recomendados para melhor performance).

***Espaço em Disco:*** Mínimo de 20 GB livres.

***Conexão à Internet***: Necessária para instalar dependências e atualizações.

* **Passos para instalar o software necessário:**

1. ***Instalação do .NET Core SDK*:**
   * Acesse o site oficial do .NET Core: [dotnet.microsoft.com/download](https://dotnet.microsoft.com/download).
   * Baixe e instale a versão 6.0 ou superior compatível com seu sistema operacional.
   * Verifique a instalação abrindo o terminal e executando dotnet --version.
2. ***Instalação do Node.js e NPM:***
   * Acesse o site oficial do Node.js: [nodejs.org](https://nodejs.org/).
   * Baixe e instale a versão LTS (Long Term Support) recomendada.
   * Verifique a instalação executando node -v e npm -v no terminal.
3. ***Instalação do SQL Server:***
   * Acesse o site oficial da Microsoft: [microsoft.com/sql-server](https://www.microsoft.com/sql-server).
   * Baixe a versão Developer ou Express do SQL Server.
   * Siga o assistente de instalação e configure uma instância local.
4. ***Instalação de Visual Studio Code/Visual Studio:***
   * Acesse o site oficial do Visual Studio Code: [code.visualstudio.com](https://code.visualstudio.com/) ou do Visual Studio: [visualstudio.microsoft.com](https://visualstudio.microsoft.com/).
   * Baixe e instale a versão mais recente.
5. ***Instalação do Git:***
   * Acesse o site oficial do Git: [git-scm.com](https://git-scm.com/).
   * Baixe e instale a versão mais recente para o seu sistema operacional.
   * Verifique a instalação com git --version no terminal.

* **Passos para configurar o ambiente de desenvolvimento:**

1. ***Clone o Repositório do Projeto:***
   * No terminal, navegue até o diretório onde deseja clonar o projeto.
   * Execute git clone <URL-do-repositório> para clonar o projeto.
2. ***Instale as Dependências Back-end:***
   * Navegue até a pasta do projeto utilizando cd <nome-do-projeto>.
   * Execute dotnet restore para instalar todas as dependências do .NET Core.
3. ***Instale as Dependências Front-end:***
   * Navegue até a pasta client ou equivalente.
   * Execute npm install para instalar as dependências do React.js.
4. ***Configuração do Banco de Dados:***
   * No SQL Server, crie um banco de dados novo ou restaure um backup, conforme necessário.
   * Atualize a string de conexão no arquivo de configuração (appsettings.json) com os detalhes da sua instância do SQL Server.
5. ***Executar o Projeto:***
   * Para o back-end, execute dotnet run na pasta do projeto.
   * Para o front-end, navegue até a pasta client e execute npm start.

* **Dependências**

**Dependências do Back-end (.NET Core):**

***Entity Framework Core***: ORM para comunicação com o banco de dados.

***Swashbuckle.AspNetCore:*** Para documentação de API com Swagger.

***Microsoft.Extensions.Configuration***: Para gerenciar configurações de ambiente.

***Microsoft.IdentityModel.Tokens:*** Para autenticação JWT.

**Dependências do Front-end (React.js):**

***react-router-dom:*** Para roteamento de páginas.

***axios:*** Para realizar requisições HTTP.

***react-bootstrap:*** Componentes visuais baseados em Bootstrap.

***react-icons:*** Para ícones SVG.

***formik*** *e* ***yup:*** Para validação de formulários.

**Dependências de Desenvolvimento:**

***eslint:*** Para padronização e linting de código.

***prettier:*** Para formatação de código.

***jest:*** Para testes unitários no front-end.

***xUnit:*** Para testes unitários no back-end.

**Desenvolvimento**A estrutura do projeto é organizada de maneira modular para facilitar o desenvolvimento, a manutenção e a escalabilidade do sistema. Cada pasta principal tem uma função específica no ciclo de vida do desenvolvimento e tentei organizá-las conforme boas práticas recomendadas para aplicações baseadas em **.NET Core** no back-end e **React.js** no front-end. A seguir, está a descrição das principais pastas e sua função no projeto:

* **No Back-end**



O **PortalBackend** segue uma arquitetura em camadas, organizada de forma clara para separar as responsabilidades, mantendo o sistema modular e fácil de manter. Cada camada tem uma função específica, como o gerenciamento da lógica de apresentação, regras de negócio, acesso a dados e serviços. Abaixo estão os detalhes de cada camada e seus componentes.

#### **1. Camada de Apresentação (Controllers)**

A camada de apresentação é responsável por lidar com as requisições HTTP, controlando a comunicação entre o front-end e o back-end. Essa camada contém os **Controllers**, que são responsáveis por receber as requisições dos clientes, processá-las e devolver as respostas.

* **/Controllers**
  + **/UsuarioController.cs**: Controlador principal para gerenciar as operações relacionadas aos usuários. Ele expõe as rotas da API, como cadastro, login, atualização de senha, etc., e interage com as outras camadas para processar essas solicitações.

#### **2. Camada de Serviços (Aplicação)**

A camada de **Serviços** (ou Aplicação) contém a lógica de negócios e as interfaces que definem as operações que podem ser realizadas no sistema. Nessa camada, o foco é a implementação dos casos de uso da aplicação, como criação e autenticação de usuários.

* **/Aplicacao**
  + **/Interface**
    - **IAplicacaoUsuario.cs**: Interface que define as operações relacionadas aos usuários, como criação, autenticação e atualização. Ela serve como um contrato que outras classes devem implementar para garantir que a lógica de negócio seja respeitada.

#### **3. Camada de Domínio**

A camada de **Domínio** contém as regras de negócio e as entidades que representam os objetos principais do sistema. Essa camada é essencial para definir a lógica de negócios do sistema, sem se preocupar com a interface ou o acesso aos dados.

* **/Dominio**
  + **/Entidades.cs**: Contém as definições gerais de entidades usadas no sistema.
  + **/Usuario.cs**: Classe que representa a entidade **Usuário** no sistema, com suas propriedades como nome, e-mail, senha, etc.

#### **4. Camada de Repositório**

A camada de **Repositório** é responsável pelo acesso a dados. Ela contém as implementações que conectam o sistema ao banco de dados, além de gerenciar as operações CRUD (Create, Read, Update, Delete) para as entidades do domínio.

* **/Repositorio**
  + **/Configuracoes**
    - **UsuarioConfiguracoes.cs**: Contém as configurações específicas para a entidade **Usuário** no repositório, como mapeamentos de banco de dados.
  + **/Contexto**
    - **Contexto.cs**: Representa o contexto do banco de dados utilizado pelo Entity Framework. Gerencia a conexão com o banco de dados e o mapeamento entre as entidades e as tabelas.
  + **/Interface**
    - **IRepositorioUsuario.cs**: Interface que define os métodos para acessar os dados relacionados a **Usuários**, como salvar, editar e buscar usuários no banco de dados.

#### **5. Modelos de Requisição e Resposta**

A pasta **/Models** organiza as classes que representam as requisições e respostas da API, fornecendo uma estrutura clara para a troca de informações entre o sistema e o cliente.

* **/Models**
  + **/Requisicao**
    - **UsuarioAtualizarSenha.cs**: Modelo que representa a requisição de atualização de senha do usuário, contendo os campos necessários para essa operação.
    - **UsuarioAtualizar.cs**: Modelo para a atualização dos dados do usuário.
    - **UsuarioAutenticar.cs**: Modelo usado no processo de autenticação (login) de usuários.
    - **UsuarioCriar.cs**: Representa a requisição para criar um novo usuário.
    - **UsuarioDetalhes.cs**: Usado para consultar os detalhes de um usuário específico.
  + **/Resposta**
    - **UsuarioLogado.cs**: Contém as informações de um usuário após o login.

**UsuarioResposta.cs**: Modelo padrão de resposta para operações relacionadas aos usuários, como sucesso ou erro.

* **No front-end**



O sistema é organizado em diversas telas e componentes, seguindo uma estrutura modular, facilitando a manutenção e a expansão da aplicação. Abaixo está uma descrição detalhada das principais pastas e arquivos.

***/src:*** Esta pasta é a raiz do código-fonte da aplicação. Dentro dela estão organizados os componentes principais e serviços que compõem a interface e a lógica de negócios da aplicação.

* **/Services**: Contém os serviços responsáveis por interagir com as APIs e outras fontes de dados externas. Esses serviços são usados em toda a aplicação para executar operações como o cadastro, edição de usuários e recuperação de dados. Aqui está um exemplo de serviço:
  + **apiServices.js**: Implementa a comunicação com as APIs que gerenciam as operações do sistema, como autenticação de usuários, atualização de dados e outros.
* **/Componentes**: Armazena componentes reutilizáveis que podem ser usados em diversas telas do sistema. Esses componentes são pequenos blocos de interface, responsáveis por partes da visualização e lógica do front-end.
* **/Telas**: Esta pasta contém as telas principais da aplicação, ou seja, as páginas ou views que os usuários finais irão acessar. Cada subpasta representa uma tela ou funcionalidade específica do sistema:
  + **/Cadastrar**: Tela usada para o cadastro de novos usuários no sistema.
  + **/EditarUsuario**: Tela que permite a edição dos dados dos usuários já cadastrados.
  + **/Holerites**: Exibe os holerites dos funcionários, permitindo que eles visualizem ou façam download de seus comprovantes de pagamento.
  + **/Home**: A página principal, que funciona como um painel ou dashboard, onde os usuários têm acesso rápido às funcionalidades disponíveis.
  + **/Login**: Tela responsável pela autenticação dos usuários, onde eles inserem suas credenciais para acessar o sistema.
  + **/Menu**: Exibe o menu principal do portal do funcionário, com as opções de navegação para outras telas, como banco de horas, editar perfil, etc.
  + **/PerfilUsuario**: Tela que exibe as informações do perfil do usuário, permitindo que ele visualize e edite seus dados pessoais. Obs.: A ideia inicial era que tivesse uma tela de perfil, porém o projeto foi alterado e essa tela não foi usada, mas não foi deletada para possíveis implementações futuras.
  + **/RestaurarUsuario**: Tela onde o usuário pode restaurar sua conta ou reativar um perfil que foi desativado anteriormente.
* **/App.cs**: Arquivo de configuração geral do back-end do sistema (se aplicável), responsável por conectar a lógica de negócios com a interface.
* **/App.js**: O componente raiz da aplicação, responsável por renderizar a interface e gerenciar a navegação entre as diferentes telas do sistema. Este arquivo é o ponto de entrada do front-end React.
* **/Index.js**: Arquivo principal de inicialização da aplicação. Aqui, a aplicação é renderizada no DOM, e configurações importantes, como a inicialização da árvore de componentes do React, são realizadas.

**Documentação da API**

Abaixo está a descrição detalhada dos principais endpoints da API, incluindo os métodos HTTP, parâmetros e exemplos de requisição e resposta.

***Endpoint: HealthCheck***

Verifica se a API está funcionando corretamente.

**URL**: GET/HealthCheck

**Descrição**: Verifica a conexão com o servidor.

**Resposta**:

Status 200 (OK): {"status": "Conexao bem-sucedida"}

***Endpoint: Obter Usuário por ID***

Obtém os detalhes de um usuário específico com base no seu ID.

**URL**: GET/Obter/{usuarioId}

**Parâmetro**: usuarioId (int) - ID do usuário.

**Exemplo de Requisição**:

**Exemplo de Respostas**:

Status 200 (OK): 

Status 400 (Bad Request): {"message": "Usuário não encontrado"}

#### **Endpoint: Listar Usuários**

Lista todos os usuários, filtrando por status (ativos ou inativos).

**URL**: GET /Listar

**Parâmetro**: ativo (bool) - Filtra por status de usuário (ativo/inativo).

**Exemplo de Requisição:**

**Resposta**:

Status 200 (OK)



***Endpoint: Cadastrar Usuário***

Cadastra um novo usuário.

**URL**: POST /Cadastrar

**Body**:

UsuarioCriar (JSON):

**Exemplo de requisição:**

**Respostas**:

Status 200 (OK):

Status 400 (Bad Request): {"message": "Erro ao cadastrar o usuário"}

#### **Endpoint: Atualizar Usuário**

Atualiza os dados de um usuário existente.

**URL**: PUT /Atualizar/{id}

**Body**:

UsuarioAtualizar (JSON):

**Exemplo de Requisição**:

**Respostas**:

Status 200 (OK): {} (sucesso)

Status 400 (Bad Request): {"message": "Erro ao atualizar o usuário"}

***Endpoint: Logar Usuário***

Autentica um usuário com base em CPF e senha.

**URL**: POST /Logar

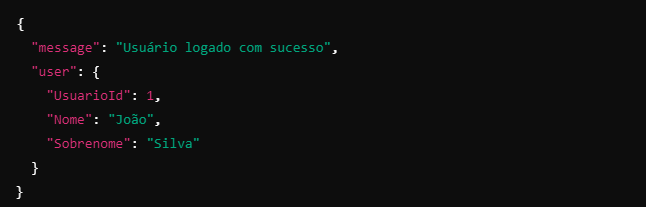
**Método HTTP:**

**Body**:

UsuarioAutenticar (JSON)

**Exemplo de Requisição:**

**Respostas**:

Status 200 (OK):

Status 401 (Unauthorized): {"message": "CPF ou senha incorretos"}

***Endpoint: Deletar Usuário***

Deleta um usuário por ID. Fazendo com que o status *ativo* do usuário, se torne *inativo.*

**URL**: DELETE /Delete/{usuarioId}

**Parâmetro**: usuarioId (int) - ID do usuário.

**Exemplo de Requisição:**

**Respostas**:

Status 200 (OK): {} (sucesso)

Status 400 (Bad Request): {"message": "Erro ao deletar o usuário"}

***Endpoint: Restaurar Usuário***

Restaura um usuário previamente deletado. Fazendo que que o status *inativo* do usuário, se torne *ativo*

**URL**: PUT /Restaurar/{usuarioId}

**Parâmetro**: usuarioId (int) - ID do usuário.

**Exemplo de Requisição**:

**Respostas**:

Status 200 (OK): {} (sucesso)

Status 400 (Bad Request): {"message": "Erro ao restaurar o usuário"}

**Interface do Usuário**

**• Descrição das Funcionalidades da Interface: Explicação das principais funcionalidades da interface do usuário.**

**TELAS e EXPLICAÇÃO:**

**Banco de Dados**

Apesar de saber da necessidade da criação de uma tabela para tratar da entidade *Endereço*. Para não correr o risco de me complicar e atrasar a entrega do projeto, parti para uma abordagem bem simples e utilizando somente uma tabela. Assim o endereço é um campo da tabela principal (e única) do projeto. Mudanças e melhorias neste sentido serão feitas no futuro.

**Considerações finais**

* **Lições aprendidas**

Fazer este projeto foi algo desafiador e prazeroso ao mesmo tempo. A cada obstáculo vencido, a cada dúvida sanada, a cada erro corrigido e cada nova ideia, passos eram dados na direção da minha evolução pessoal e profissional. Poderia ter sido melhor com certeza, mas tenho a sensação de tarefa cumprida, uma sensação de orgulho por ver realizado aquilo que idealizei.

Em alguns momentos achei que não conseguiria concluir o projeto, uma sensação de impotência ao me deparar com um problema que parecia sem solução, as vezes algo simples, mas que impedia o funcionamento correto do programa. Mas eu devia perseverar, insistir, pesquisar, não tinha outra saída, era entregar o projeto ou entregar o projeto em minha mente.

A satisfação e alegria quando disse “Terminei!” foram indescritíveis. Ver tudo funcionando exatamente como planejado, como desejado é algo incrível. Espero poder fazer parte de muitos mais projetos como este em minha vida, estar sempre aprendendo mais, criando mais para poder facilitar a vida de pessoas com projetos inteligentes é satisfatório e recompensador.

* **Melhores práticas**

Durante o desenvolvimento do projeto, algumas boas práticas foram adotadas para garantir a qualidade, manutenção e segurança do sistema. Essas práticas cobrem desde a *arquitetura do sistema* até o *desenvolvimento do código* e *processos ágeis.*

Para garantir uma arquitetura limpa, as responsabilidades foram separadas em camadas distintas, o que facilita a manutenção, escalabilidade e teste unitários. A lógica de negócios está isolada na camada de *Domínio*, por exemplo.

Na codificação teve-se a preocupação de deixar o *código limpo*, utilizando nomes de variáveis, métodos e classes descritivos e claros, além da separação lógica em pequenos métodos que realizam uma única tarefa. Além disso houve a preocupação com tratamento de exceções, onde foram implementados controles de erros e exceções, garantindo respostas apropriadas e amigáveis aos usuários.

O *Swagger* foi utilizado para verificação da API, facilitando a integração com outros sistemas e garantindo que todos os endpoints estejam bem descritos. Além de utilização de ***HTTPS*** para garantir uma comunicação mais segura entre clientes e o servidor.

Por se tratar de um projeto individual, a metodologia *Scrum*, não foi amplamente aplicada, porém, ao longo do projeto certas metas eram traçadas afim de manter um ritmo constante de entrega e organização. E ainda houve a preocupação com o uso de padrões de **versionamento de API** para suportar mudanças sem comprometer a compatibilidade com versões anteriores.

Para gerenciamento das dependências foram utilizados o *NuGet* e o *npm,* o primeiro para as bibliotecas *.NET* e o segundo para o frontent em *JavaScript.* Garantindo assim que todas as bibliotecas necessárias sejam versionadas e facilmente instaladas, garantindo que o ambiente de desenvolvimento seja replicável.

Adotar essas práticas garante um **código robusto**, **escalável** e que pode ser facilmente mantido e ampliado por outros desenvolvedores no futuro.

* **Próximos passos**
  + Escalabilidade e Desempenho: Adicionar uma tabela para tratar da entidade *Endereço* visando a otimização do banco de dados, visando melhorar o desempenho em cenários de grande volume de dados.
  + Melhoria da Interface de Usuário (UI): Refinar o design da interface com foco em uma experiência mais intuitiva, utilizando componentes mais modernos e responsivos.
  + Segurança: Adicionar um *token* ao login de usuário como uma segunda etapa de autenticação. Afim de garantir mais segurança e integridade das informações.
  + Integrações: Explorar integrações com APIs de terceiros, para ampliar a funcionalidade do sistema.
  + Novas funcionalidades: Além de garantir a implementação das funcionalidades já previstas no projeto, a implementação de novas funcionalidades se faz necessária, como por exemplo, disparo de mensagens automáticas ou não aos usuários.

**Anexos**

* **Referencias e recursos adicionais**
  + Documentação oficial do .NET: <https://docs.microsoft.com/dotnet/>
  + Documentação do React: <https://reactjs.org/docs/getting-started.html>
  + Documentação do SQL Server: <https://learn.microsoft.com/sql/>
  + Swagger API Documentation: <https://swagger.io/docs/>
  + Documentação do React Icons: <https://react-icons.github.io/react-icons/>
  + ChatGPT OpenAI: <https://openai.com/chatgpt/>
  + Documentação do Bootstrap: <https://getbootstrap.com/docs/>
* **Links úteis**
  + GitHub do projeto: <https://github.com/nomeDoProjeto>
  + Google: <https://www.google.com.br>
  + YouTube: <https://youtube.com>
* **Créditos e agradecimentos**

Agradeço a todos que fizeram parte da primeira turma da Itera360. Pessoas maravilhosas que tive o prazer de conhecer e conviver por estes meses. Devo agradecer todos meus colegas que tiraram um pouco do seu tempo para me ajudar, mas devo agradecimentos especiais a alguns, que em vários momentos me ajudaram quando tive dificuldades e me incentivaram quando achei que não conseguiria. Agradeço de coração ao Gabriel que, devo dizer, teve muita paciência comigo durante o curso. Agradeço ao Anderson, que se mostrava empolgado para sanar alguma dúvida, ao Paulo, sempre disposto a ajudar e com uma didática incrível. Agradeço também ao Marcos, um cara fenomenal e extremamente capaz e pôr fim ao Felipe, o idealizador deste projeto, que por conta dele, tive o privilégio de estar com pessoas tão maravilhosas e me evoluir pessoal e profissionalmente. Não posso dizer quais caminhos me esperam, mas é certo que o percorrerei com intensidade e paixão, com coragem e fibra e que por todo sempre as pessoas possam ter a oportunidade que tive, de melhorar, de crescer e de conhecer.