Sumário

Documentação API - Grupo GTI

Estrutura Básica da API desenvolvida pelo Grupo GTI

O que é uma API:

A API, que significa Interface de Programação de Aplicações (em inglês, Application Programming Interface), pode ser definida como conjunto de regras que permite a comunicação de diferentes softwares (com diferentes linguagens na maioria das vezes) entre si. Consequentemente, há a definição das formas de como os componentes de software devem interagir, permitindo que desenvolvedores acessem e utilizem funcionalidades específicas de um sistema, serviço ou aplicação.

Em um contexto de desenvolvimento web, uma API ou microsserviço geralmente refere-se a um conjunto de endpoints (URLs) e métodos (como GET, POST, PUT, DELETE) que permitem que os clientes (aplicações ou serviços) interajam com o servidor para realizar operações específicas, como receber dados, enviar dados e até mesmo atualizar informações.

No contexto do projeto essa comunicação por meio de endpoints será feita, mas também há possibilidade da inclusão de outros microsserviços que podem alimentar algum tipo de necessidade diante do escopo e inferência de inteligência a partir de seções específicas da estrutura de dados.

Modelo MVC

Junto ao desenvolvimento de APIs há muitos caminhos de padronização a se seguir. A partir de convenção de mercado, por meio de boas práticas da utilização de ASP.NET em C#, e padronização de arquitetura, estamos seguindo o MVC. O Model-View-Controller (MVC) é um padrão arquitetural que separa a aplicação em três componentes principais: Model, View e Controller. O objetivo é organizar o código-fonte de maneira mais clara, facilitando a manutenção e a escalabilidade do software.

Como foi dito anteriormente, no contexto do .NET Core e C#, o MVC é frequentemente utilizado para o desenvolvimento de aplicações web, incluindo APIs.

Aqui está uma breve explicação de cada componente do padrão MVC:

- Model: O Model representa os dados e a lógica de negócios da aplicação; Ele lida com a manipulação, validação e armazenamento de dados (definição de variáveis de estado); O Model também pode notificar a View sobre alterações nos dados usando o padrão Observer ou outras técnicas.
- View: A View é responsável por exibir os dados e interagir com o usuário, ela recebe dados do Model e os apresenta de maneira apropriada para o usuário. Na arquitetura web, a View é geralmente implementada usando tecnologias como HTML, CSS e JavaScript.
- Controller: O Controller age como intermediário entre a View e o Model, ele processa as entradas do usuário provenientes da View, atualiza o Model de acordo e também manipula as requisições HTTP e decide qual View deve ser renderizada. PS: Contêm métodos que são mapeados para os diversos verbos HTTP (GET, POST, PUT, DELETE) e roteados para responder a determinados endpoints

Na implementação do MVC em qualquer projeto ASP.NET , os controladores são classes que herdam da classe Controller. Cada método público em um controlador representa uma ação que pode ser acessada via HTTP. A estrutura básica de uma ação de controlador geralmente inclui a lógica para processar uma requisição e retornar uma resposta adequada.

Principais Funcionalidades da API:

- **Roteamento:**
  - O roteamento em ASP.NET mapeia URLs de solicitação para métodos de controlador específicos.
- **Injeção de Dependência:**
  - O ASP.NET Core suporta injeção de dependência nativamente.
  - Isso facilita a organização e teste do código, permitindo a injeção de serviços necessários em controladores e outros componentes.
- **Serialização/Deserialização:**
  - O ASP.NET Core lida automaticamente com a serialização de objetos para JSON e vice-versa, facilitando a comunicação com clientes que consomem dados em formato JSON.
- **Swagger/OpenAPI:**
  - A integração do Swagger ou OpenAPI permite a geração automática de documentação para a API, facilitando o entendimento e a interação com os endpoints.
- **Atributos de Roteamento e Validação de Modelos:**
  - A utilização de atributos de roteamento (`[Route]`) nos controladores e a validação de modelos (`[Required]`, `[MaxLength]`, etc.) tornam o código mais expressivo e seguro.

- **Autenticação e Autorização:**
    - O ASP.NET oferece suporte a autenticação e autorização integradas. Você pode configurar esquemas de autenticação, como JWT, e definir políticas de autorização.
- **Middleware para Tratamento de Erros:**
    - Middleware de tratamento de erros permite lidar com exceções globalmente, retornando respostas adequadas para diferentes situações.
- **Suporte Multiplataforma:**
    - O .NET é multiplataforma, suportando Windows, Linux e macOS.

Detalhamento por Tabelas

Empregados

EmployeeService (Camada de Serviço)

A classe `EmployeeService` é projetada para lidar com lógica de negócios relacionada a dados de funcionários. Implementa a interface `IEmployeeService`, definindo métodos para recuperar funcionários individuais, uma lista de funcionários e espaços reservados para operações como inserção, atualização e exclusão de funcionários.

Principais pontos:

- **Injeção de Dependência:** O serviço recebe uma implementação de `IEmployeeRepository` por meio de seu construtor. Isso promove um acoplamento flexível e facilita a troca ou extensão da camada de acesso a dados sem modificar o serviço.
- **Abstração:** O serviço abstrai os detalhes do acesso aos dados, contando com os métodos definidos na interface `IEmployeeRepository`. Essa abstração ajuda a manter uma clara separação entre a lógica de negócios e as preocupações de acesso aos dados.
- **Consistência:** O serviço adere a um design de API consistente, implementando os métodos definidos na interface `IEmployeeService`. Essa consistência é crucial para que outras partes da aplicação, como os controladores, interajam de maneira integrada com o serviço.

Interface IEmployeeService

A interface IEmployeeService define um contrato para serviços relacionados a dados de funcionários na aplicação. Esta interface declara métodos que encapsulam operações específicas de manipulação e consulta de dados de funcionários. Ela desempenha um papel vital na separação de responsabilidades entre a camada de serviço e a camada de repositório, garantindo uma abstração clara e consistente para as operações relacionadas a funcionários.

Métodos:

- GetEmployeeById(int id):
    - Retorna um único funcionário com base no seu identificador único.
    - Parâmetro id: O identificador único do funcionário.
    - Retorna uma tarefa assíncrona que encapsula o modelo do funcionário.
- GetEmployees():

    - Retorna uma coleção de todos os funcionários disponíveis.
        - Retorna uma tarefa assíncrona que encapsula uma coleção de modelos de funcionários.
- InsertEmployee(EmployeeModel employeeModel):
    - Insere um novo funcionário na fonte de dados.
    - Parâmetro employeeModel: O modelo de dados do novo funcionário a ser inserido.
    - Retorna uma tarefa assíncrona que encapsula o número de linhas afetadas pela operação de inserção.
- DeleteEmployee(int id):
    - Exclui um funcionário com base no seu identificador único.
    - Parâmetro id: O identificador único do funcionário a ser excluído.
    - Retorna uma tarefa assíncrona que encapsula o número de linhas afetadas pela operação de exclusão.

Uso:

Padrão de Serviço: A interface IEmployeeService é implementada pela classe EmployeeService, que atua como a camada de serviço para operações relacionadas a funcionários. Isso facilita a substituição de implementações de serviço sem afetar as partes da aplicação que dependem dessa interface.

Consistência na API: A interface estabelece um contrato consistente para serviços relacionados a funcionários. Isso facilita a previsibilidade e o entendimento das operações que podem ser realizadas por meio da camada de serviço.

Abstração de Operações: Os métodos na interface abstraem as operações comuns relacionadas a funcionários, como obtenção de dados individuais e de lista, inserção e exclusão. Isso promove uma abordagem limpa e bem definida para a manipulação de dados de funcionários na aplicação.

EmployeeRepository (Camada de Repositório)

A classe `EmployeeRepository` é responsável por interagir com o banco de dados para recuperar dados de funcionários. Implementa a interface `IEmployeeRepository`, definindo métodos para recuperar funcionários individuais, uma lista de funcionários e espaços reservados para operações como inserção, atualização e exclusão de funcionários.

Recursos-chave do repositório:

- **Interação com o Banco de Dados:** O repositório utiliza o Dapper, um micro-ORM, para interagir com um banco de dados PostgreSQL. O Dapper simplifica o processo de mapeamento de registros do banco de dados para objetos C#.
- **Operações Assíncronas:** Todas as operações de banco de dados são assíncronas (palavras-chave `async` e `await`) para garantir que a aplicação permaneça responsiva, especialmente em cenários com várias solicitações sendo processadas simultaneamente.
- **Consultas Parametrizadas:** As consultas SQL incluem parâmetros para evitar vulnerabilidades de injeção de SQL. As consultas recuperam colunas específicas da tabela `empregados` com base no ID do funcionário ou recuperam um conjunto limitado de registros para a lista de funcionários.
- **Injeção de Construtor:** O repositório recebe uma string de conexão com o banco de dados como parâmetro em seu construtor. Essa abordagem permite flexibilidade na conexão com diferentes bancos de dados e promove a testabilidade.

EmployeeModel.cs

O arquivo `EmployeeModel.cs` contém a definição da classe `EmployeeModel`, que atua como um modelo de dados representando os atributos associados a um funcionário. O propósito dessa classe é encapsular e estruturar os dados relacionados aos funcionários, facilitando seu uso na aplicação.

Propriedades:

- **Id:** Representa o identificador único de um funcionário.
- **NPessoal:** Indica o número pessoal de um funcionário.
- **SgEmp:** Representa o código ou identificador do funcionário.
- **TextoRh:** Armazena o texto relacionado a RH associado ao funcionário.
- **CentroCst:** Representa o código do centro de custo de um funcionário.
- **CentroCusto:** Descreve o centro de custo associado ao funcionário.
- **Cargo:** Especifica o cargo ou posição do funcionário.
- **DataNascimento:** Representa a data de nascimento do funcionário.

Uso:

Esta classe é utilizada como uma estrutura padronizada para representar dados de funcionários em toda a aplicação. Instâncias de `EmployeeModel` são usadas para transmitir informações entre diferentes camadas da aplicação, como entre a camada de serviço e a camada de repositório.

IEmployeeRepository.cs

O arquivo `IEmployeeRepository.cs` define a interface `IEmployeeRepository`, delineando o contrato que qualquer classe concreta de repositório de funcionários deve seguir. Essa interface serve como um modelo para interagir com dados de funcionários armazenados em uma fonte de dados, promovendo uma abordagem consistente e intercambiável para acesso aos dados.

Métodos:

- **GetEmployees:** Recupera uma lista de todos os funcionários da fonte de dados.
- **GetEmployeeById:** Recupera um funcionário específico pelo seu identificador único na fonte de dados.
- **InsertEmployee:** Insere um novo funcionário na fonte de dados.
- **DeleteEmployee:** Exclui um funcionário da fonte de dados com base em seu identificador único.
- **UpdateEmployee:** Atualiza um funcionário existente na fonte de dados.

Uso:

- **Injeção de Dependência:** A interface `IEmployeeRepository` é utilizada na classe `EmployeeService`, permitindo que o serviço interaja com a fonte de dados sem estar fortemente acoplado a uma implementação específica. Isso promove flexibilidade e testabilidade.
- **Design de API Consistente:** Ao aderir a esta interface, qualquer implementação concreta do repositório deve fornecer esses métodos padrão, garantindo uma API consistente e previsível para trabalhar com dados de funcionários.

- **Abstração:** A interface abstrai os detalhes de como os dados são armazenados ou recuperados, proporcionando uma clara separação de preocupações entre a camada de serviço e a camada de repositório. Essa abstração facilita alterações na camada de acesso aos dados sem afetar a lógica de negócios na camada de serviço.

Plano de Saúde - ps_amil

1. HealthInsurance Model:

A camada de modelo `HealthInsuranceModel` contém a definição da classe que representa os dados relacionados à Saúde Suplementar. Esta classe, `HealthInsuranceModel`, encapsula os atributos associados a um plano de saúde específico, fornecendo uma estrutura padronizada para manipulação desses dados.

Propriedades:

- **Id:** Identificador único do plano de saúde.
- **NomeCliente:** Nome do cliente associado ao plano.
- **NomeBanco:** Nome do banco relacionado ao plano.
- **NomeProduto:** Nome do produto do plano.
- **CodigoPessoa:** Código da pessoa relacionada ao plano.
- **NomePaciente:** Nome do paciente vinculado ao plano.
- **CodigoPaciente:** Código único do paciente.
- **Matricula:** Número de matrícula do paciente no plano.
- **Cpf:** CPF do paciente.
- **DataNascimento:** Data de nascimento do paciente.
- **Sexo:** Gênero do paciente.
- **Parentesco:** Relação de parentesco do paciente com o titular do plano.
- **CodigoPlano:** Código do plano de saúde.
- **NomePlano:** Nome do plano de saúde.
- **GrupoFamiliar:** Número que identifica o grupo familiar associado ao plano.
- **DataEvento:** Data do evento relacionado ao plano.
- **CodigoProcedimentoFinal:** Código do procedimento final.
- **DescricaoProcedimentoFinal:** Descrição do procedimento final.
- **CodigoServico:** Código do serviço associado ao plano.
- **DescricaoServico:** Descrição do serviço associado ao plano.
- **TipoEventoCbhpm:** Tipo de evento segundo a CBHPM (Classificação Brasileira Hierarquizada de Procedimentos Médicos).

- **TipoEventoFinal:** Tipo de evento final.
- **TipoUtilizacao:** Tipo de utilização do plano.
- **NumeroSubFatura:** Número da subfatura associada ao plano.
- **NumeroContrato:** Número do contrato do plano.
- **CodigoPrestador:** Código do prestador de serviços.
- **NomePrestador:** Nome do prestador de serviços.
- **Especialidade:** Especialidade do prestador de serviços.
- **Sinistro:** Informações sobre sinistro.
- **NumeroEvento:** Número do evento associado ao plano.
- **Planta:** Informações sobre a planta relacionada ao plano.

Recursos Chave:

- Padronização de Dados: A classe HealthInsuranceModel estabelece uma estrutura consistente para representar informações de planos de saúde, promovendo a uniformidade e facilitando o entendimento dos dados.

2. HealthInsuranceRepository:

A camada de repositório `HealthInsuranceRepository` é responsável por interagir com o banco de dados para recuperar dados relacionados à Saúde Suplementar. Essa classe implementa a interface `IHealthInsuranceRepository` e utiliza a biblioteca Dapper para simplificar as operações de mapeamento dos resultados das consultas para objetos C#.

Métodos:

- **`GetHealthInsuranceById(int id): HealthInsuranceModel`**
  - Recupera um plano de saúde com base no seu identificador único.
- **`GetHealthInsurances(): IEnumerable<HealthInsuranceModel>`**
  - Recupera uma lista de planos de saúde.
- **`GetPlanByCodigoPaciente(int CodigoPaciente): string`**
  - Recupera o plano associado a um código de paciente.
- **`GetProcedures(): IEnumerable<HealthInsuranceModel>`**
  - Recupera uma lista de procedimentos relacionados aos planos de saúde.

Recursos Chave:

- Conexão com o Banco de Dados: A classe HealthInsuranceRepository gerencia a interação com o banco de dados PostgreSQL, fornecendo métodos para recuperar dados de planos de saúde e procedimentos associados.
- Mapeamento de Dados: Utilizando o Dapper, realiza o mapeamento eficiente dos resultados das consultas para objetos HealthInsuranceModel, simplificando a manipulação de dados.

3. IHealthInsuranceRepository:

A interface `IHealthInsuranceRepository` define um contrato que qualquer repositório concreto de Saúde Suplementar deve seguir. Ela fornece uma abstração para a camada de serviço, definindo métodos padrão para interagir com dados relacionados à Saúde Suplementar.

Métodos:

- **`GetHealthInsurances(): IEnumerable<HealthInsuranceModel>`**
    - Recupera uma lista de planos de saúde.
- **`GetHealthInsuranceById(int id): HealthInsuranceModel`**
    - Recupera um plano de saúde com base no seu identificador único.
- **`GetPlanByCodigoPaciente(int CodigoPaciente): string`**
    - Recupera o plano associado a um código de paciente.
- **`GetProcedures(): IEnumerable<HealthInsuranceModel>`**
    - Recupera uma lista de procedimentos relacionados aos planos de saúde.

Recursos Chave:

- Abstração de Dados: A interface abstrai a lógica de acesso aos dados, proporcionando uma padronização nos métodos que devem ser implementados pelos repositórios concretos.

4. HealthInsuranceService:

A camada de serviço `HealthInsuranceService` implementa a interface `IHealthInsuranceService` e é responsável por conter a lógica de negócios relacionada à Saúde Suplementar. Essa camada utiliza o repositório `IHealthInsuranceRepository` para interagir com os dados e fornecer operações de alto nível para outras partes do sistema.

Métodos:

- **`GetHealthInsuranceById(int id): HealthInsuranceModel`**
    - Recupera um plano de saúde com base no seu identificador único.
- **`GetHealthInsurances(): IEnumerable<HealthInsuranceModel>`**
    - Recupera uma lista de planos de saúde.
- **`GetPlanByCodigoPaciente(int CodigoPaciente): string`**
    - Recupera o plano associado a um código de paciente.
- **`GetProcedures(): IEnumerable<HealthInsuranceModel>`**
    - Recupera uma lista de procedimentos relacionados aos planos de saúde.

Recursos Chave:

- Lógica de Negócios: Implementa a lógica de negócios relacionada aos planos de saúde, fornecendo métodos para recuperar informações específicas e gerenciar operações relacionadas à Saúde Suplementar.

- Abstração de Repositório: Utiliza a abstração proporcionada pela interface IHealthInsuranceRepository, permitindo que diferentes implementações de repositório possam ser injetadas.

5. IHealthInsuranceService:

A interface `IHealthInsuranceService` define um contrato para os serviços relacionados à Saúde Suplementar. Essa interface estabelece métodos padrão que devem ser implementados pela classe de serviço concreta.

Métodos:

- **`GetHealthInsurances(): IEnumerable<HealthInsuranceModel>`**
  - Recupera uma lista de planos de saúde.
- **`GetHealthInsuranceById(int id): HealthInsuranceModel`**
  - Recupera um plano de saúde com base no seu identificador único.
- **`GetPlanByCodigoPaciente(int CodigoPaciente): string`**
  - Recupera o plano associado a um código de paciente.
- **`GetProcedures(): IEnumerable<HealthInsuranceModel>`**
  - Recupera uma lista de procedimentos relacionados aos planos de saúde.

Recursos Chave:

- Abstração de Serviços: A interface define métodos padronizados que os serviços relacionados à Saúde Suplementar devem implementar. Facilita a injeção de dependência e promove a consistência nas operações.

CID - cid_f_2023_geral

1. CID Model:

A camada de modelo CIDModel contém a definição da classe que representa os dados relacionados ao Código Internacional de Doenças (CID). Essa classe encapsula os atributos associados a um CID específico, fornecendo uma estrutura padronizada para manipulação desses dados.

Propriedades:

- Mes: Mês associado ao CID.
- N_pessoal: Número pessoal associado ao CID.
- Atestados: Número de atestados relacionados ao CID.
- Dias: Dias associados ao CID.
- Diretoria: Diretoria associada ao CID.
- Unidade: Unidade associada ao CID.
- Genero: Gênero associado ao CID.
- Categoria: Categoria associada ao CID.

- Cid: Código Internacional de Doenças.
- DescricaoDetalhada: Descrição detalhada associada ao CID.
- DescricaoResumida: Descrição resumida associada ao CID.
- DiagnosticoAtestadoInicial: Diagnóstico relacionado ao atestado inicial.
- CausaRaiz: Causa raiz associada ao CID.
- Outros: Outras informações associadas ao CID.
- Jornada: Jornada associada ao CID.

Uso:

- Utilizado para representar e manipular dados relacionados ao CID em várias camadas do sistema.
- Passado entre camadas, como entre o serviço e o repositório, para transferência de dados.

Recursos Chave:

- Estrutura de dados padronizada para representar informações relacionadas ao CID.
- Facilita a consistência e clareza na manipulação de dados em diferentes partes do sistema.

2. CIDRepository:

A camada de repositório CIDRepository é responsável por interagir com o banco de dados para recuperar dados relacionados ao Código Internacional de Doenças (CID). Essa classe implementa a interface ICIDRepository e utiliza a biblioteca Dapper para simplificar as operações de mapeamento dos resultados das consultas para objetos C#.

Métodos:

- GetCauseByHealthUnit(string unidade): CIDModelRecupera a causa associada a uma unidade de saúde.
- GetCID(): IEnumerable Recupera uma lista de informações relacionadas ao CID.
- InsertEmployee(CIDModel cidModel): intInsere um novo registro relacionado ao CID no banco de dados.
- UpdateEmployee(CIDModel cidModel): intAtualiza um registro relacionado ao CID no banco de dados.

Uso:

- Utilizado pela camada de serviço para acessar dados relacionados ao CID no banco de dados.
- Responsável pela consulta e retorno de informações específicas sobre o CID.

Recursos Chave:

- Integração com o banco de dados para recuperar dados relacionados ao CID.
- Utilização do Dapper para mapeamento eficiente dos resultados das consultas para objetos C#.

3. ICIDRepository:

A interface ICIDRepository define um contrato que qualquer repositório concreto de CID deve seguir. Ela fornece uma abstração para a camada de serviço, definindo métodos padrão para interagir com dados relacionados ao Código Internacional de Doenças.

Métodos:

- GetCID(): Recupera uma lista de informações relacionadas ao CID.
- GetCauseByHealthUnit(string unidade): Recupera a causa associada a uma unidade de saúde.
- InsertEmployee(CIDModel cidModel): Insere um novo registro relacionado ao CID no banco de dados.
- UpdateEmployee(CIDModel cidModel): Atualiza um registro relacionado ao CID no banco de dados.

Uso:

- Implementado pelos repositórios concretos para garantir consistência nos métodos de acesso a dados.
- Utilizado pela camada de serviço para acessar dados relacionados ao CID.

Recursos Chave:

- Contrato que estabelece métodos padrão para interação com dados relacionados ao CID.
- Promove consistência na implementação dos métodos pelos repositórios concretos.

4. CID Service & ICID Service:

A camada de serviço CIDService implementa a interface ICIDService e é responsável por conter a lógica de negócios relacionada ao Código Internacional de Doenças (CID). Essa camada utiliza o ICIDRepository para acessar e manipular dados relacionados ao CID.

Métodos:

- GetCauseByHealthUnit(string unidade): Obtém a causa associada a uma unidade de saúde.
- GetCID(): Obtém uma lista de informações relacionadas ao CID.
- InsertEmployee(CIDModel cidModel): Insere um novo registro relacionado ao CID no banco de dados.
- UpdateEmployee(CIDModel cidModel): Atualiza um registro relacionado ao CID no banco de dados.

Uso:

- Utilizado pelos controladores e outras partes do sistema para acessar operações relacionadas ao Código Internacional de Doenças.
- Contém a lógica de negócios para manipulação de dados e tomada de decisões.

Recursos Chave:

- Encapsula a lógica de negócios relacionada ao Código Internacional de Doenças.
- Fornece uma interface de alto nível para acessar e manipular dados de CID.

Zenklub

1. Zenklub Model:

A camada de modelo `ZenklubModel` contém a definição da classe que representa os dados relacionados ao Zenklub. Essa classe encapsula os atributos associados a um registro Zenklub específico, fornecendo uma estrutura padronizada para manipulação desses dados.

Propriedades:

- **Periodo:** Período associado ao registro Zenklub.
- **Mes:** Mês associado ao registro Zenklub.
- **Nome:** Nome associado ao registro Zenklub.
- **NPessoal:** Número pessoal associado ao registro Zenklub.
- **CodigoValidacao:** Código de validação associado ao registro Zenklub.
- **Departamento:** Departamento associado ao registro Zenklub.
- **TotalSessoes:** Total de sessões associadas ao registro Zenklub.

Uso:

- Utilizado para representar e manipular dados relacionados ao Zenklub em várias camadas do sistema.
- Passado entre camadas, como entre o serviço e o repositório, para transferência de dados.

Recursos Chave:

- Estrutura de dados padronizada para representar informações relacionadas ao Zenklub.
- Facilita a consistência e clareza na manipulação de dados em diferentes partes do sistema.

2. ZenklubRepository:

A camada de repositório `ZenklubRepository` é responsável por interagir com o banco de dados para recuperar dados relacionados ao Zenklub. Essa classe implementa a interface `IZenklubRepository` e utiliza a biblioteca Dapper para simplificar as operações de mapeamento dos resultados das consultas para objetos C#.

Métodos:

- **`GetDepartment(): IEnumerable<ZenklubModel>`**
  - Recupera uma lista de informações relacionadas ao Zenklub, incluindo período, mês, nome, número pessoal, código de validação, departamento e total de sessões.
- **`GetTotalSessionsByDepartment(): ZenklubModel`**
  - Recupera informações sobre o total de sessões agrupadas por departamento no Zenklub.

Uso:

- Utilizado pela camada de serviço para acessar dados relacionados ao Zenklub no banco de dados.
- Responsável pela consulta e retorno de informações específicas sobre o Zenklub.

Recursos Chave:

- Integração com o banco de dados para recuperar dados relacionados ao Zenklub.
- Utilização do Dapper para mapeamento eficiente dos resultados das consultas para objetos C#.

3. IZenklubRepository:

A interface `IZenklubRepository` define um contrato que qualquer repositório concreto de Zenklub deve seguir. Ela fornece uma abstração para a camada de serviço, definindo métodos padrão para interagir com dados relacionados ao Zenklub.

Métodos:

- **`GetDepartment(): IEnumerable<ZenklubModel>`**
    - Recupera uma lista de informações relacionadas ao Zenklub, incluindo período, mês, nome, número pessoal, código de validação, departamento e total de sessões.
- **`GetTotalSessionsByDepartment(): ZenklubModel`**
    - Recupera informações sobre o total de sessões agrupadas por departamento no Zenklub.

Uso:

- Implementado pelos repositórios concretos para garantir consistência nos métodos de acesso a dados.
- Utilizado pela camada de serviço para acessar dados relacionados ao Zenklub.

Recursos Chave:

- Contrato que estabelece métodos padrão para interação com dados relacionados ao Zenklub.
- Promove consistência na implementação dos métodos pelos repositórios concretos.

4. IZenklubService:

A interface `IZenklubService` define um contrato que qualquer serviço concreto de Zenklub deve seguir. Ela fornece uma abstração para outras partes do sistema, definindo métodos padrão para acessar operações relacionadas ao Zenklub.

Métodos:

- **`GetDepartment(): IEnumerable<ZenklubModel>`**
    - Obtém uma lista de informações relacionadas ao Zenklub, incluindo período, mês, nome, número pessoal, código de validação, departamento e total de sessões.
- **`GetTotalSessionsByDepartment(): ZenklubModel`**
    - Obtém informações sobre o total de sessões agrupadas por departamento no Zenklub.

Uso:

- Implementado pelos serviços concretos para fornecer uma interface de alto nível para acessar e manipular dados de Zenklub.
- Utilizado por outras partes do sistema, como controladores, para acessar operações relacionadas ao Zenklub.

Recursos Chave:

- Fornecer uma interface consistente para acessar operações relacionadas ao Zenklub em níveis mais altos do sistema.
- Promover a modularidade e flexibilidade, permitindo a troca fácil de implementações concretas.

5. IZenklubService (Continuação):

Métodos:

- **`GetDepartment(): IEnumerable<ZenklubModel>`**
    - Obtém uma lista de informações relacionadas ao Zenklub, incluindo período, mês, nome, número pessoal, código de validação, departamento e total de sessões.
- **`GetTotalSessionsByDepartment(): ZenklubModel`**
    - Obtém informações sobre o total de sessões agrupadas por departamento no Zenklub.

Uso:

- Implementado pelos serviços concretos para fornecer uma interface de alto nível para acessar e manipular dados de Zenklub.
- Utilizado por outras partes do sistema, como controladores, para acessar operações relacionadas ao Zenklub.

Recursos Chave:

- Fornecer uma interface consistente para acessar operações relacionadas ao Zenklub em níveis mais altos do sistema.
- Promover a modularidade e flexibilidade, permitindo a troca fácil de implementações concretas.

6. ZenklubService:

A classe `ZenklubService` é a implementação concreta da interface `IZenklubService`. Ela lida com a lógica de negócios relacionada ao Zenklub, utilizando o repositório correspondente para acessar e manipular os dados.

Métodos:

- **`GetDepartment(): Task<IEnumerable<ZenklubModel>>`**
  - Chama o método correspondente no repositório para obter informações sobre departamentos no Zenklub.
- **`GetTotalSessionsByDepartment(): Task<ZenklubModel>`**
  - Chama o método correspondente no repositório para obter informações sobre o total de sessões agrupadas por departamento no Zenklub.

Uso:

- Utilizado pelos controladores ou outras partes do sistema que necessitam de operações relacionadas ao Zenklub.
- Contém a lógica de negócios específica do Zenklub, mantendo uma separação clara entre a lógica de negócios e o acesso aos dados.

Recursos Chave:

- Implementação concreta dos métodos definidos na interface `IZenklubService`.
- Facilita a integração entre a lógica de negócios e a camada de acesso aos dados.
- Promove a modularidade e testabilidade do código.

Implementação de Serviço de Log

Visão Geral

O serviço de log é uma parte essencial de qualquer aplicativo, pois registra eventos e informações importantes que ocorrem durante a execução do aplicativo. Ele permite rastrear problemas, diagnosticar falhas e analisar o comportamento do sistema.

Nesta implementação, utilizamos o framework Serilog para realizar o logging de terminal, e em algumas coisas no de service. O Serilog é uma biblioteca de logging altamente configurável e flexível que oferece suporte a diversos provedores de log e formatos de saída.

Componentes Principais

No desenvolvimento desta etapa, realizamos o log de terminal e o de service.

Log de Terminal

Na implementação do log de terminal, utilizamos o Serilog como biblioteca no intuito de registrar eventos e informações importantes durante a execução do programa. Ele fornece uma maneira de monitorar o comportamento do sistema, identificar problemas e diagnosticar falhas. Neste sentido, fizemos a implementação em cada um dos endpoints de nosso projeto. A lógica implementada segue os seguintes padrões e blocos:

`try-catch`

O bloco `try-catch` é uma estrutura de controle em C# que permite capturar exceções durante a execução do código. Ele tenta executar o código dentro do bloco `try` e, se ocorrer uma exceção, o controle é transferido para o bloco `catch` para tratamento.

```
try
{
    // Código que pode gerar exceções
}
catch (Exception ex)
{
    // Tratamento da exceção
}
```

Log Information

O método Log.Information é usado para registrar uma mensagem informativa no log. Ele é útil para indicar eventos importantes que ocorrem durante a execução do programa.

Exemplo de implementação no nosso projeto:

```
Log.Information("POST request received for {desired_endpoint} endpoint.");
```

Chamada do Serviço de Log

Neste caso, o serviço de log a partir de uma variável pré-definida é "chamado" para registrar uma entrada de log.

Exemplo de como foi implementado:

```
await {_pre_defined_variable}.LogEntry({model_or_anyparameter});
```

Exception - Tratamento de Exceções

No bloco "catch", exceções são capturadas e tratadas. Como exemplo para ilustração, aqui o método "Log.Error" é usado para registrar uma mensagem de erro no log, juntamente com detalhes sobre a exceção que ocorreu:

```
catch (Exception ex)
{
    Log.Error(ex, "An error occurred while processing POST request for LogEntry endpoint.");
    return StatusCode(500, "An error occurred while processing your request.");
}
```

Sendo assim, na prática o sistema vai seguir este método de sequência (exemplo caso o sistema flagre um erro):

a. O programa tenta executar o código dentro do bloco try.
b. Se uma exceção ocorrer durante a execução do código, o controle é transferido para o bloco catch.
c. Uma mensagem de erro é registrada no log usando Log.Error.
d. O método retorna um código de status HTTP 500 para indicar que ocorreu um erro durante o processamento da solicitação.

Exemplo do log de terminal rodando em nossa aplicação:

Log de Service para Regras de Negócios

O log de serviço captura e registra eventos mais específicos, e denotados importantes durante a execução do projeto. Nesta implementação específica, o log de serviço foi definido para realizar uma operação de inserção (POST) no banco de dados por meio de funções que vão direcionar as informações que serão devolvidas. Essa operação de inserção é realizada com base nas variáveis definidas no modelo de log (LogModel), que contém os dados a serem registrados no banco de dados.

É importante salientar que para criar sua estrutura seguimos os padrões do Modelo MVC, como foi feito para a implementação geral dos serviços do projetos.

LogModel

Este modelo define a estrutura dos dados que serão registrados no log. Ele inclui variáveis como Id_usuario, Status_Code, Classe, Message, Exception e Data_Hora, que representam informações relevantes sobre o evento registrado.

```
public class LogModel
{
    public int Id_usuario { get; set; }
    public int Status_Code { get; set; }
    public string Classe { get; set; }
    public string Message { get; set; }
    public string Exception { get; set; }
    public DateTime Data_Hora { get; set; }
}
```

ILogRepository

O ILogRepository é uma interface que define operações para o acesso aos dados de log, como inserção de novos logs no banco de dados.

```
public interface ILogRepository
{
    Task<IEnumerable<LogModel>> DbLogging(LogModel logModel);
}
```

LogRepository

Já o LogRepository é uma classe que implementa a interface ILogRepository. Ela realiza a inserção de logs no banco de dados a partir das informações que temos da tabela.

Exemplo de como a implementação e lógica é feita:

```
public class LogRepository : ILogRepository
{
    private readonly string _dbConfig;

    public LogRepository(string dbConfig)
    {
        _dbConfig = dbConfig;
    }

    public async Task<IEnumerable<LogModel>> DbLogging(LogModel logModel)
    {
        // Implementação da inserção de log no banco de dados
    }
}
```

ILogService

O ILogService é uma interface que define operações para o serviço de log, como registrar um novo log e recuperar logs existentes.

Sua implementação é feita desta forma:

```
public interface ILogService
{
    Task LogEntry(LogModel logModel);
    IEnumerable<LogModel> GetHardcodedLogEntries();
    Task<IEnumerable<string>> GetLogColumnValues(LogModel logModel);
}
```

LogsService

O LogsService é uma classe que implementa a interface ILogService. Ela contém a lógica para registrar logs e recuperar informações de log.

Aqui usamos alguns dados para preencher o banco de dados de acordo com as informações que gostaríamos de receber:

```
public class LogsService : ILogService
{
    private readonly ILogRepository _logRepository;

    public LogsService(ILogRepository logRepository)
    {
        _logRepository = logRepository ?? throw new ArgumentNullException(nameof(logRepository));
    }

    public async Task LogEntry(LogModel logModel)
    {
        try
        {
            await _logRepository.DbLogging(logModel);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error logging data: {ex.Message}");
            throw;
        }
    }

    public IEnumerable<LogModel> GetHardcodedLogEntries()
    {
        var logEntries = new List<LogModel>
        {
            new LogModel
            {
                Id_usuario = 1,
                Status_Code = 200,
                Classe = "ClasseTeste",
                Message = "Recebido com sucesso",
                Exception = "",
                Data_Hora = DateTime.Now
            },
            new LogModel
            {
                Id_usuario = 2,
                Status_Code = 404,
                Classe = "ClasseTesteErrado",
                Message = "Error message",
                Exception = "Exception: Error while processing data; Generic error",
                Data_Hora = DateTime.Now
            }
        };

        return logEntries;
    }
```

LogsController

O LogsController é um controlador responsável por expor endpoints relacionados ao serviço de log.

Exemplo de estrutura de sua implementação:

```
[Route("api/logs")]
[ApiController]
public class LogsController : ControllerBase
{
    private readonly ILogService _logService;

    public LogsController(ILogService logService)
    {
        // Seguir definições de variáveis e métodos do service
    }

    [HttpPost]
    public async Task<IActionResult> LogEntry([FromBody] LogModel logModel)
    {
        // Lógica de Implementação do endpoint para registrar um novo log
    }
}
```

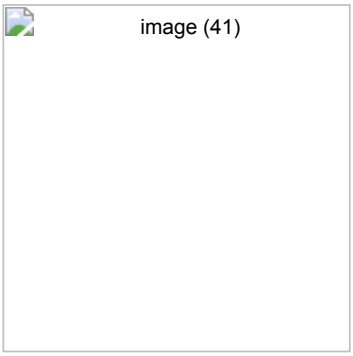
Configuração Final para Visualização

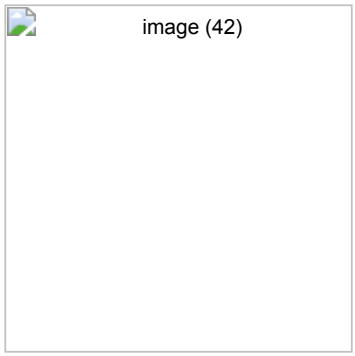
No arquivo Program.cs, a configuração do serviço de log é feita durante a configuração do contêiner de injeção de dependência:

```
builder.Services.AddScoped<ILogService, LogsService>();
builder.Services.AddScoped<ILogRepository>(_ => new LogRepository(configuration["DB_CONFIG"]));
```

Isso registra a implementação LogsService da interface ILogService e o LogRepository como implementação do ILogRepository no contêiner de injeção de dependência.

Exemplo de log das regras de negócio em funcionamento:





Integração do Grafana com PostgreSQL para Monitoramento dos Logs de Negócios

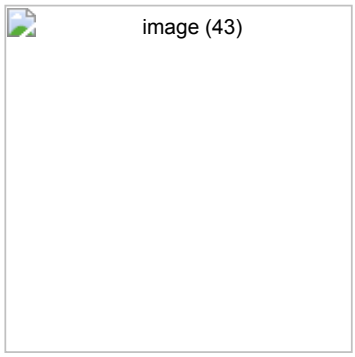
Grafana

O Grafana é uma ferramenta avançada para visualização de dados que permite criar painéis dinâmicos utilizando dados de diversas fontes. A integração pode ser feita de várias formas, mas nesta ocasião utilizamos um bancos de dados PostgreSQL para a visualização em tempo real de dados de logs da nossa aplicação..

Configuração da Fonte de Dados

Para conectar o Grafana ao PostgreSQL, é necessário configurar o PostgreSQL como uma fonte de dados dentro do Grafana. Isso envolve fornecer os detalhes de conexão do banco de dados e ajustar as permissões adequadas, das quais configuramos.

Visualização do Dashboard



Painel de Monitoramento

Usuários Ativos na Aplicação

- **Tipo**: Número único destacado.

- **Objetivo**: Exibir o número atual de usuários ativos.
- **Consulta SQL**: `SELECT COUNT(DISTINCT user_id) FROM sessions WHERE active = true;`

Status Code Time Horizon

- **Tipo**: Gráfico de linhas.
- **Objetivo**: Mostrar a frequência de códigos de status HTTP ao longo do tempo.
- **Consulta SQL**: `SELECT data_hora, status_code, COUNT(*) FROM http_requests GROUP BY data_hora, status_code ORDER BY data_hora;`
- **Eixos**:
  - X: Horário (`data_hora`)
  - Y: Frequência dos códigos de status (`COUNT dos status_code`)

Logs na Aplicação

- **Tipo**: Tabela.
- **Objetivo**: Listar mensagens de log recentes.
- **Colunas**:
  - `id_usuario`: ID do usuário.
  - `message`: Mensagem de log.
  - `classe`: Classificação da mensagem.
  - `data_hora`: Timestamp do log.

Processo de Deploy com GitHub Actions e Docker

Este documento fornece uma visão geral do processo de deploy do backend em C# do nosso projeto, utilizando um pipeline de Integração Contínua (CI) e Entrega Contínua (CD) implementado com GitHub Actions com containerização no Docker.

GitHub Actions Workflow

O arquivo YAML no diretório `.github` define o pipeline CI/CD e é dividido em três principais trabalhos: `build`, `test` e `deploy`.

Build Job

```
jobs:
  build:
    name: "Build"
    runs-on: ubuntu-latest

    steps:
    - name: "Checkout"
      uses: actions/checkout@v2

    - name: "Set Dotnet"
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: '8.0.x'

    - name: "Restore dependencies"
      run: dotnet restore

    - name: "Build api"
      run: dotnet build --configuration Release
```

Processo realizado no **build**:

- Checkout: Obtém o código-fonte do repositório para o runner do GitHub.
- Set Dotnet: Configura o ambiente com a versão especificada do .NET 8.
- Restore dependencies: Instala todas as dependências do .NET.
- Build: Compila o o backend .NET para produção.

Test Job

```
jobs:
  test:
   name: Unit Test
   runs-on: ubuntu-latest
   needs: build
   steps:
    - name: "Checkout"
      uses: actions/checkout@v2

    - name: "Set Dotnet"
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: '8.0.x'

    - name: "Restore dependencies"
      run: dotnet restore

    - name: "Run unit test"
      run: dotnet test --no-build --verbosity normal
```

Processo adicional realizado no "Test Job":

- needs: Especifica que o trabalho de teste depende da conclusão bem-sucedida do trabalho de construção.
- Run unit test: A adição de teste aqui ela é feita a partir dos arquivos de teste na pasta Tests.

Deploy Job

```
jobs:
  deploy:
    name: Deploy
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to production
        uses: johnbeynon/render-deploy-action@v0.0.8
        with:
           service-id: ${{ secrets.RENDER_SERVICE_ID }}
           api-key: ${{ secrets.RENDER_API_KEY }}
```

Processo adicional para o deploy final:

- Deploy to Production: Este passo usa uma action do GitHub para implantar o backend na plataforma especificada, neste caso, o Render.
- secrets: Utiliza secrets do repositório para autenticar no serviço de hospedagem (chaves de acesso do Render).

Docker Configuration

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 8080
EXPOSE 8081

# Usando a imagem SDK do .NET para construir o aplicativo
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src

# Copiando o arquivo do projeto e restaurando dependências
COPY ["GTI.Dashboard.WebApi.csproj", "."]
RUN dotnet restore "GTI.Dashboard.WebApi.csproj"

# Copiando todo o código e construindo o aplicativo
COPY . .
WORKDIR "/src"
RUN dotnet build "GTI.Dashboard.WebApi.csproj" -c $BUILD_CONFIGURATION -o /app/build

# Publicando o aplicativo
FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "GTI.Dashboard.WebApi.csproj" -c $BUILD_CONFIGURATION -o /app/publish

# Usando a imagem ASP.NET para executar o aplicativo publicado
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "GTI.Dashboard.WebApi.dll"]
```

A configuração do Docker constrói uma imagem contendo a estrutura .NET:

- Base: Define a imagem base para o aplicativo, com a versão do .NET 8 e as portas 8080 e 8081 expostas.
- Build: Copia o código-fonte do aplicativo, restaura as dependências e compila o aplicativo.
- Publish: Publica o aplicativo compilado em um diretório específico.
- Final: Copia o aplicativo publicado para a imagem final e define o ponto de entrada para executar o aplicativo.

CI/CD Pipeline Aplicado no projeto

O pipeline de CI/CD aplicado funciona da seguinte forma:

- Integração Contínua (CI): No push para o branch principal ou em um pull request, o GitHub Actions executa o build para compilar o código e executar os testes unitários (test).

- Entrega Contínua (CD): Se o build e testes passarem, o trabalho deploy é executado para implantar o aplicativo compilado em um ambiente de produção.

Este pipeline garante que cada alteração no código seja automaticamente construída e testada antes de ser entregue ao ambiente de produção, minimizando os riscos associados ao processo de deploy manual e aumentando a eficiência do ciclo de vida do desenvolvimento de software.

Documentação dos Testes de Unidade para CIDService

Este arquivo contém um pequeno relatório dos testes de unidade para o serviço CIDService, usando o framework NUnit e o pacote Moq para mockar dependências. O foco está em testar a lógica de negócio associada ao gerenciamento de CID (Classificação Internacional de Doenças) dentro de um contexto corporativo.

Dependências

- NUnit
- Moq

Classe `CIDServiceTests`

A classe `CIDServiceTests` inclui testes para duas funções principais dentro do serviço `CIDService`. As dependências são mockadas utilizando Moq, permitindo testes isolados da lógica de negócio.

Configuração (*Setup*)

Antes de cada teste, um ambiente é configurado onde:

- Um mock do repositório ICIDRepository é criado.
- Uma instância do CIDService é criada com o repositório mockado.

Funções Testadas

1. `Should_GetCIDWithSuccess()` Testa a função `GetCID()` do serviço, que deve buscar informações de CID com sucesso.

- **Arrange**: Configura o repositório mockado para retornar uma lista esperada de objetos `CIDModel`.
- **Act**: Chama a função `GetCID()` do serviço.
- **Assert**: Verifica se o resultado corresponde ao esperado, tanto em quantidade quanto em valores específicos.

2. `Should_GetCauseByHealthUnitWithSuccess()` Testa a função `GetCauseByHealthUnit()` do serviço, que deve buscar a causa raiz de atestados por unidade de saúde com sucesso.

- **Arrange**: Configura o repositório mockado para retornar uma lista esperada de objetos `CIDModel` com base em uma unidade de saúde específica.
- **Act**: Chama a função `GetCauseByHealthUnit()` do serviço com uma unidade de saúde como parâmetro.
- **Assert**: Verifica se o resultado é conforme o esperado, checando tanto a contagem de itens retornados quanto os valores específicos de unidade de saúde e causa raiz.

Execução dos Testes

Para executar esses testes, garanta que as dependências estão corretamente instaladas e configure seu ambiente de teste para utilizar o NUnit. Utilize um runner de testes de sua preferência que suporte NUnit para executar os casos de teste definidos em `CIDServiceTests`.

Outra opção é utilizar o comando `dotnet test --no-build --verbosity normal` no diretório do projeto para executar todos os testes de unidade definidos no projeto.

Para abrir o terminal é só usar o atalho `Ctrl + "`.

Conclusão

Os testes unitários detalhados no arquivo `UnitTest.cs` são essenciais para garantir a robustez e a fiabilidade do serviço `CIDService`, que lida com a gestão de informações relativas à Classificação Internacional de Doenças (CID) em um contexto corporativo. Utilizando NUnit e Moq para simular dependências, estes testes focam em validar a lógica de negócio sem a necessidade de integrar com sistemas externos ou a base de dados, proporcionando uma maneira eficaz de identificar e corrigir erros precocemente no ciclo de desenvolvimento.

Próximos Passos

Para expandir a cobertura de testes e continuar a aumentar a confiabilidade do software, considere os seguintes passos para os demais arquivos e funcionalidades do projeto:

1. **Expandir Cobertura de Testes**: Identifique áreas do código ou funcionalidades críticas que ainda não estão cobertas por testes unitários. Priorize testes para esses caminhos críticos para garantir uma base de código mais segura.
2. **Testes de Integração**: Além dos testes unitários, implemente testes de integração para validar as interações entre diferentes componentes ou serviços, especialmente aquelas que envolvem chamadas externas a APIs ou acesso a bases de dados.
3. **Refatoração com TDD (Test-Driven Development)**: Para novas funcionalidades ou ao refatorar partes existentes do sistema, considere usar o TDD. Escreva os testes antes do código de produção para guiar o design da sua implementação e garantir que a nova funcionalidade esteja corretamente testada desde o início.
4. **Melhoria Contínua**: Use os resultados dos testes para refinar e melhorar continuamente o código. Isso inclui não apenas a correção de bugs, mas também a otimização da lógica de negócios e a melhoria da performance e da segurança do sistema.

Fontes e Recursos utilizados no desenvolvimento da documentação:

OpenAI. (2023). GPT-3.5 - A Powerful Language Model for Natural Language Processing. Disponível em: https://www.openai.com/ (https://www.openai.com/). Acesso em 03/03/2024.

BillWagner. (2023, February 15). Conceitos de programação (C#). Microsoft Learn. Disponível em: https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/ (https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/) . Acesso em 10/03/2024

BillWagner. (n.d.). Documentação do .NET. Microsoft Learn. https://learn.microsoft.com/pt-br/dotnet/ (https://learn.microsoft.com/pt-br/dotnet/) . Acesso em 10/03/2024

Devmedia. (n.d.). Guia de C#. DevMedia. https://www.devmedia.com.br/guia/linguagem-csharp/38152 (https://www.devmedia.com.br/guia/linguagem-csharp/38152) . Acesso em 15/03/2024