Manual de Arquitetura da API

“Sistema de Tarefas”

Índice

---

1. Visão Geral do Sistema

Este sistema é um portfólio profissional para Gestão de Tarefas, baseado em conceitos avançados e práticas modernas de segurança, arquitetura e controle de acesso.

1.1 Entidades

O sistema trabalha com a seguinte estrutura:

- **Modelos de Tarefa**: Templates para criação de tarefas, contendo sequência de trâmites.
- **Modelos de Trâmite**: Passos que compõem um modelo de tarefa, podendo ter usuários associados para execução e revisão.
- **Tarefas**: Instâncias criadas a partir do modelo, com trâmites automáticos criados e vinculados.
- **Trâmites**: Passos reais na execução da tarefa, com status, usuários e notas.
- **Usuários**: Com níveis de permissão variados, que controlam o acesso e ações possíveis.

1.2 Configuração do banco de dados SQL Server e Token.

A configuração deve ser feita no arquivo **appsettings.json** que já contém um modelo de exemplo. O arquivo além das configurações de Token de acesso ao Banco, também permite configurar por

exemplo a expiração de senha dos usuários do sistema em “ExpiracaoSenhaDias” ou até mesmo se deseja disponibilizar o Swagger "ExibirSwagger": true.

---

2. Entidades e Relacionamentos

2.1. Usuário

- **Id**: Identificador único (PK).
- **Login**: Nome de login, usado para autenticação.
- **Nome**: Nome completo ou social, usado para exibição.
- **Nível**: Controle de acesso (0 a 4, onde 0 é bloqueado).
- **Senha Hash**: Armazenamento seguro da senha.

2.2. ModeloTarefa

- Representa um template de tarefa com múltiplos modelos de trâmite.
- **Relacionamento:** Um para muitos com ModeloTramite.

2.3. ModeloTramite

- Define cada etapa de um modelo de tarefa.
- Pode conter:
    - **UsuarioTramitadorId** (FK opcional): Usuário designado para executar a etapa.
    - **UsuarioRevisorId** (FK opcional): Usuário responsável pela revisão da etapa.
- Caso esses campos sejam 0 (zero), o sistema trata como null — ou seja, sem usuário definido. A Foreign Key opcional será detalha nas Observações Técnicas.

2.4. Tarefa

- Instância concreta criada a partir de um ModeloTarefa.
- Contém status, data de criação, criador, e a lista de trâmites.

2.5. Tramite

- Instância real de um modelo trâmite ligado a uma tarefa.
- Contém status, usuário que executa, usuário revisor (se houver), notas, datas de início e fim. Também possuir TRA_USU_ID_Tramitador e TraUsuIdTramitador como campos opcionais.

---

3. Fluxo de Trabalho

3.1. Criação de Modelo de Tarefa e Trâmites

- Administrador (nível 1) cria um ModeloTarefa.
- Define sequência ordenada de ModeloTramite.

- Para cada ModeloTramite, opcionalmente define:
    - Usuário tramitador
    - Usuário revisor

3.2. Criação da Tarefa

- Usuário com nível 2 ou superior cria uma tarefa com base em um ModeloTarefa.
- O sistema cria automaticamente os trâmites para essa tarefa, com as associações de usuários conforme definido no modelo.

3.3. Execução do Trâmite

- Se o usuário tramitador está definido no modelo, este será associado automaticamente na criação da tarefa.
- Caso contrário, um usuário autorizado (nível 3) pode **assumir o trâmite** por meio do endpoint dedicado.
- Usuário deve iniciar o trâmite (**ComecarExecucaoTramite**).
- Finaliza o trâmite (**FinalizarExecucaoTramite**) e precisa inserir uma nota obrigatória.

3.4. Revisão

- Se há usuário revisor definido, ele deve aprovar ou reprovar o trâmite após finalização.
- Aprovação libera o próximo trâmite.
- Reprovação pode fazer repetir o trâmite corrente, com nota justificativa.

3.5. Conclusão da Tarefa

- Quando todos os trâmites estiverem finalizados e aprovados, a tarefa é considerada concluída.

---

4. Controle de Acesso Nível e Segurança

4.1. Níveis de Usuário

Nível	Permissões
0	Usuário bloqueado — sem acesso
1	Administrador: cria usuários, modelos, ativa/desativa tarefas, pode deletar trâmites
2	Cria tarefas, atua como revisor
3	Pode assumir e executar trâmites
4	Somente consulta (não grava nada)

4.2. Autenticação e Autorização JWT

- O sistema usa **JWT Bearer Token** para autenticação.
- Os tokens carregam claims com informações importantes do usuário:

    - "i": usuário ID criptografado
    - "n": nível de acesso criptografado
- As políticas de autorização validam esses níveis ao descriptografar os claims.
- Uso de **expiração de token** para segurança (configurado com ClockSkew e validação do tempo).

4.3. Claims e Segurança

- Todas as ações que dependem do usuário autenticado utilizam as informações criptografadas do JWT.
- Isso evita que um usuário informe outro ID manualmente em parâmetros, garantindo integridade e segurança.
- Exemplo no método Servico.Claims() que extrai usuarioID e nivelUsuario dos claims.

4.4. Middleware de Logging e Tratamento Global de Erros

- Logs detalhados são feitos com **Serilog**, armazenados em arquivos diários e console.
- Tratamento global de exceções com middleware para retornar respostas JSON consistentes.
- Logs de requisições HTTP para auditoria e debug.

---

5. Técnicas e Tecnologias Utilizadas

- **.NET 9+ e ASP.NET Core WebAPI**: arquitetura moderna e performática.
- **Entity Framework Core** com SQL Server para persistência.
- **JWT Authentication** com claims criptografados para segurança.
- **Políticas de autorização** personalizadas para níveis de acesso.
- **Serilog** para logging estruturado.
- **Response Compression** com Brotli e Gzip para melhorar performance.
- **Swagger / OpenAPI** para documentação e testes da API.
- **Automapper** para mapeamento de DTOs e entidades.
- **Middleware customizado** para:
    - Logs de requisições
    - Tratamento global de exceções
    - Respostas padronizadas
- **HTTPS com certificado PFX** configurado diretamente no Kestrel.

---

6. Observações Técnicas

- **Foreign Key** de forma Opcional. Exemplo para UsuarioTramitadorId e UsuarioRevisorId, são campos opcionais, podem ser atribuído valor 0 (zero), indicando “sem usuário definido”, não utilizando este campo. No código, isso é tratado como null. Entretanto se optar por informar, preserva a **integridade referencial** com a entidade Usuários, não permitindo a exclusão do usuário.
- Tokens possuem tempo de expiração controlado, reforçando segurança.
- Política de autorização via RequireAssertion permitindo validar regras complexas via claims descriptografados.
- Uso de JsonIgnoreCondition.WhenWritingNull para omitir dados nulos na serialização JSON.
- Resposta HTTP estruturada para indicar sucesso, falha e códigos específicos (ResponseCode enum).

---

7. Sugestões para Completar ou Expandir

- Modelos e endpoints para **reset de senha** e **expiração de senha**.
- Log detalhado de ações do usuário para auditoria.
- Testes automatizados unitários e de integração.
- Interface frontend (Angular, React, Blazor) consumindo API.
- Documentação mais detalhada do contrato da API (ex.: exemplos de requests/responses).
- Workflow visual para os trâmites e estados das tarefas.
- Política de refresh tokens para JWT.

---

8. Exemplo de Uso do Método Claims

Este método assegura que o sistema sempre utilize o usuário correto r o nível de acesso que está autenticado no token, reforçando a segurança e integridade dos dados.

Método:
```
public static (int usuarioID, int nivelUsuario) Claims()
```
Forma de uso:
```
retorno = await
_tramitesRepositorio.FinalizarExecucaoTramite(tramiteNotaRequest.IdTramite
, tramiteNotaRequest.Nota.Trim(), Servico.Claims().usuarioID);
```

9. Padrão de Resposta da API

Todas as respostas da API são baseadas em um modelo que implementa a interface IResponseModel. Esse padrão ajuda a manter consistência e clareza na comunicação entre backend e frontend ou consumidores da API.

9.1. Interface IResponseModel

```
public interface IResponseModel
{
    string RM { get; set; }
    ResponseCode RC { get; set; }
    string errorCode { get; set; }
    bool OK { get; set; }
}
```

- **RM**: Mensagem que descreve o resultado da operação, útil para feedback para usuário ou logs.
- **RC**: Enum customizado que categoriza o tipo de resposta (exemplo: sucesso, erro, validação, exceção).
- **ErrorCode**: Erro para facilitar tratamento de mensagens pelo frontend, inclusive para tradução se desejar.
- **OK**: Booleano que indica se a operação foi bem-sucedida (true) ou falhou (false).

9.2. Uso prático das respostas

Classes que implementam IResponseModel para retornar objetos padronizados. Toda classe Response deste projeto deriva da interface IresponseModel.

9.3. Funções helper para retorno

Na controller é usado métodos para retornar essas respostas com status HTTP adequados, ex:

```
return Controladores.Retorno(this, retorno, ResponseCode.BadRequest, "Parâmetro incorreto.");
```

Aqui, o método Retorno (ou similar) encapsula o padrão para enviar:

- HTTP Status correto (400, 200, 403 etc)
- Objeto que implementa IResponseModel
- Mensagem e código customizado para melhor entendimento da chamada

---

9.4. Vantagens desse padrão

- **Consistência:** Todos os endpoints retornam dados no mesmo formato.
- **Tratamento centralizado:** Facilita logging, análise e frontend interpretar resultados.
- **Extensibilidade:** Permite adicionar campos extras, ex: dados, listas, metadados.
- **Separação de responsabilidades:** Camada de API trata formatação e status, enquanto serviços retornam dados simples.

Centralização do Retorno:

Para garantir que todas as respostas sigam este padrão, foi criada uma função centralizada (a qual deve chamar como Controladores.Retorno(...)). Essa função:

- Recebe a instância de IResponseModel.
- Recebe o código de resposta e a mensagem (quando necessário).
- Configura o status HTTP apropriado e monta o corpo da resposta JSON padronizado.

Dessa forma, os controllers ficam mais limpos, e o comportamento do retorno é consistente em toda a aplicação.

9.5. Exemplo de Uso no Controller

```
        [Authorize(Policy = "NivelAcesso1a3")]
        [HttpPost("assumir-tramite/{idTramite}")]
        public async Task<ActionResult<ResponseModel>> AssumirTramite([FromRoute] int idTramite)
        {
            try
            {
                ResponseModel retorno = new();

                if (idTramite < 1)
                {
                    return Controladores.Retorno(this, retorno, ResponseCode.BadRequest,
"Parâmetro incorreto.");// Passando o tratamento direto na chamada do método  Retorno
                }

                retorno = await _tramitesRepositorio.AssumirTramite(idTramite,
Servico.Claims().usuarioID);

                return Controladores.Retorno(this, retorno);// irá tratar de acordo com o estado
do objeto retorno após uso do método  AssumirTramite, tratando corretamente o sucesso ou erro.
            }
            catch (Exception ex)
            {
                Servico.GravaLog($"{nameof(TramitesController)}.{nameof(AssumirTramite)}", ex);
                return Controladores.Retorno(this, new ResponseModel
                {
                    RM = Servico.MSG_EXCEPTION,
                    RC = ResponseCode.Excecao,
                    OK = false
                });
            }
        }
```

Benefícios desta abordagem

- **Consistência:** Todas as respostas seguem um formato previsível e estruturado.
- **Centralização:** Qualquer ajuste no padrão de resposta pode ser feito num único ponto.
- **Clareza:** Mensagens descritivas ajudam tanto desenvolvedores quanto clientes da API.
- **Manutenção facilitada:** Menos repetição de código e menos risco de inconsistências.

10. Enumeração de Códigos de Resposta e Extensões

Para tornar a comunicação da API mais clara e semântica, foi desenvolvido um enum ResponseCode que representa os principais status da API, cada um com uma descrição detalhada via atributo [Description].

Isso ajuda a:

- Centralizar os códigos de status usados na API.
- Garantir mensagens padronizadas para os retornos.
- Facilitar a manutenção e expansão futura.
- Permitir que frontends ou documentação consumam o significado dos códigos sem decoreba.

10.1. Definição do Enum ResponseCode

```
public enum ResponseCode
{
    [Description("Nulo")]
    Nulo = 0,
    [Description("Sucesso!")]
    OK = 200,
    [Description("Cadastrado com Sucesso.")]
    CadastradoSucesso = 201,
    [Description("Requisição aceita, mas ainda não processada.")]
    AceitoParaProcessamento = 202,
    [Description("Sucesso sem conteúdo útil, mas corpo base presente.")]
    SucessoSemConteudo = 204,
    [Description("Múltiplos status.")]
    MultiStatus = 207,
    [Description("Requisição inválida.")]
    BadRequest = 400,
    [Description("Não autorizado (ex: token ausente/inválido).")]
    NaoAutorizado = 401,
    [Description("Forbid - Acesso Negado.")]
    ForbidAcessoNegado = 403,
    [Description("Registro não encontrado.")]
    RegistroNaoEncontrado = 404,
    [Description("Conflito de estado (violação de regra ou integridade).")]
    Conflito = 409,
    [Description("Entidade Não Processável.")]
    EntidadeNaoProcessavel = 422,
    [Description("Muitas requisições.")]
    MuitasRequisicoes = 429,
    [Description("Exceção.")]
    Excecao = 500,
    [Description("Serviço indisponível.")]
    ServicoIndisponivel = 503,

    #region Demais Código

    [Description("Método HTTP não permitido para este recurso")]
    MetodoNaoPermitido = 405,
    [Description("Tipo de mídia não suportado")]
    TipoMidiaNaoSuportado = 415,
    [Description("Pré-condição falhou")]
    PreCondicaoFalhou = 412,
    [Description("Requisição expirou (timeout do cliente)")]
    TimeoutRequisicao = 408,

    #endregion
```

```
    }
```

10.2. Método de extensão para obter a descrição

Para acessar as descrições amigáveis do enum, implementado um método de extensão:

```
    public static class EnumExtensions
    {
        public static string GetDescription(this Enum value)
        {
            FieldInfo? field = value.GetType().GetField(value.ToString());

            if (field != null)
            {
                DescriptionAttribute? attribute =
                    Attribute.GetCustomAttribute(field, typeof(DescriptionAttribute)) as
DescriptionAttribute;

                if (attribute != null)
                {
                    return attribute.Description;
                }
            }

            return value.ToString();
        }

        public static string GetDescription(ResponseCode codigo)
        {
            FieldInfo? field = codigo.GetType().GetField(codigo.ToString());

            if (field != null)
            {
                DescriptionAttribute? attribute =
                    Attribute.GetCustomAttribute(field, typeof(DescriptionAttribute)) as
DescriptionAttribute;

                if (attribute != null)
                {
                    return attribute.Description;
                }
            }

            return codigo.ToString();
        }
    }
```

Isso permite usar, por exemplo, ResponseCode.BadRequest.GetDescription() para obter a mensagem "Requisição inválida.".

- A sobrecarga também permite chamar diretamente por ResponseCode, facilitando o uso.

10.3. Como isso é usado na API

- Ao criar respostas (IResponseModel), o enum serve para definir o código.
- A mensagem (RM) pode ser definida usando GetDescription(), garantindo que sempre haja uma mensagem coerente.
- Isso ajuda tanto no backend (logs, debugging) quanto no frontend, que pode exibir mensagens amigáveis ao usuário.

10.4. Vantagens dessa abordagem

- **Padronização:** Todas as respostas usam códigos e mensagens consistentes.
- **Manutenção:** Ao alterar a descrição de um código, isso reflete em toda API automaticamente.
- **Legibilidade:** Código mais limpo, evita "string literals" espalhados.
- **Internacionalização futura:** Essa estrutura pode ser adaptada para suportar múltiplos idiomas trocando a fonte das descrições.

10.5. Modelo Padrão de Resposta (IResponseModel) e Uso do ResponseCode

Neste sistema, todas as respostas da API seguem um padrão definido pela interface:

```
public interface IResponseModel
{
    string RM { get; set; } // Mensagem de retorno
    ResponseCode RC { get; set; } // Código enumerado do resultado da operação
    string errorCode { get; set; } // Código da mensagem para melhor tratamento
    bool OK { get; set; }  // Indica sucesso ou falha da operação
}
```

Esse modelo garante que toda resposta da API:

- Traga um código de resultado padronizado (ResponseCode), que é um enum enriquecido com descrições claras para cada situação possível (sucesso, erro, conflito, etc).
- Tenha uma mensagem legível para o cliente (RM), normalmente derivada da descrição do ResponseCode.
- Informe explicitamente se a operação foi bem sucedida (OK).

Para facilitar o uso e manter a consistência com um método centralizado para montar e retornar as respostas, que é usado em todos os controllers. Este método recebe a resposta que implementa IResponseModel e o código apropriado, e faz todo o trabalho de configurar o status HTTP, mensagem e retorno.

Exemplo simplificado do uso dentro de um endpoint:

```
retorno = await _tramitesRepositorio.AssumirTramite(idTramite, Servico.Claims().usuarioID);

return Controladores.Retorno(this, retorno);
```

Ou para respostas personalizadas, passando direto o que deseja tratar:

```
return Controladores.Retorno(this, retorno, ResponseCode.BadRequest, "Parâmetro incorreto.");
```

Essa abordagem mantém o código enxuto, evita duplicação e garante que todas as respostas sigam o mesmo padrão, melhorando a manutenção e a previsibilidade do comportamento da API.