**ALTERDATA SOFTWARE**

**S.V.R**

**DOCUMENTO DE ARQUITETURA**

**Histórico de Atualizações**

**CONTROLE DE REVISÕES DO DOCUMENTO**

|  |  |  |  |
| --- | --- | --- | --- |
| **Versão** | **Descrição** | **Data**  **dd/mm/aaaa** | **Autor** |
| 1.0 | Primeira versão do documento | 19/08/2020 | Luã Falcão |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Sumário

[1. Objetivo 4](#_Toc48773703)

[2. Sobre o SVR 4](#_Toc48773704)

[3. Web API 4](#_Toc48773705)

[4. Arquitetura 4](#_Toc48773706)

[5. Relação de Padrões de projetos e BilBiotecas 6](#_Toc48773707)

# Objetivo

Este documento tem como finalidade descrever a organização geral da API do **Sistema de Votação de Recursos** **(SVR)**, descrever a sua estrutura e padrões utilizados para organizar os seus componentes.

# Sobre o SVR

O **Sistema de Votação de Recursos** foi concebido com a necessidade de ajudar a Alterdata a priorizar as funcionalidades que mais agregam valor aos clientes através de um processo de votação, onde funcionários de todas as filiais da companhia podem acessar o sistema e votar no recurso que acreditam ser mais importante. Com isso a empresa poderá ter um controle e um mapa para ajudar a atender melhor as necessidades dos seus clientes.

# Web API

A web API é uma parte fundamental do sistema e abrange um conjunto de funcionalidades construídas sobre o framework ASP .NET Core da Microsoft e que podem ser consumidas por qualquer tipo de aplicação que faça o uso do protocolo HTTP, independente de plataforma ou linguagem de programação utilizada.

Disponibiliza endpoints para autenticar usuários, realizar o cadastro de funcionários e recursos, assim como um subsistema de votos que permite aos usuários fazer a inclusão de votações e comentários nos recursos selecionados.

Foi construída seguindo alguns princípios de REST, como o uso de interfaces uniformes para acessar recursos no servidor, o uso adequado dos verbos HTTP para interagir com os dados, um design de URIs seguindo a convenção de nomes no lugar de verbos e adoção do formato JSON para transporte de dados e serialização.

Para mais informações sobre como consumir a API acesse o seguinte endereço quando o projeto estiver em execução: [URL do Swagger aqui].

# Arquitetura

Para este projeto foi selecionado uma arquitetura multicamadas (n-tier) onde cada camada possui uma responsabilidade e representa uma abstração importante do sistema. Esta divisão em camadas garante a separação de conceitos e oferece alguns benefícios para organizar o projeto e facilitar a sua manutenção. Abaixo segue uma breve descrição das camadas da aplicação:

* Web *(Alterdata.api.Web)*

Esta camada hospeda os controles que ficam responsáveis por manipular as requisições HTTP que vem de uma aplicação externa, redireciona as solicitações para as camadas de baixo, faz o tratamento de erros e executa validações. Possui uma estrutura de controles assíncronos para garantir que as requisições que chegam ao servidor não fiquem bloqueadas caso alguma operação de I/O aconteça, como p.ex a leitura de dados do banco, a manipulação de arquivos no HD ou download de recursos oriundos de uma API externa) ajudando a garantir a responsividade e escalabilidade do sistema.

* Facade *(Alterdata.api.Facade)*

A camada Facade representa uma interface simplificada para acessar funcionalidades do subsistema que reside em camadas mais profundas da aplicação e facilita o acesso as mesmas para a camada de cima. É um padrão recomendado e bastante utilizado para envolver a camada de negócios do sistema e orquestrar operações que envolve chamada de diferentes métodos na aplicação para executar uma operação específica.

* Domínio *(Alterdata.api.Domain)*

A camada de domínio é onde reside a lógica de negócios da aplicação, ou seja, as regras do sistema que representam sua principal razão de existir e agregam valor ao negócio do cliente. Também é onde reside recursos indispensáveis para a aplicação que, se não representam completamente o negócio, faz o uso de suas entidades para oferecer funcionalidades importantes, como autenticação, validação e registro de usuários.

* Acesso a Dados *(Alterdata.api.Persistence)*

Esta é a camada de acesso a dados da aplicação, onde ficam as entidades que representam objetos do mundo real e que desejamos armazenar em nosso sistema. É o local onde ficam classes de repositório de acesso a banco de dados, entidades, DTOs, contextos e configurações de mapeamento de tabelas.

Alguns padrões de projeto utilizados nesta camada foram:

* **Repository Pattern** - permite abstrair o acesso entre as camadas de negócios e de acesso a dados e generalizar as operações para facilitar e acelerar a realização de CRUDs;
* **EF Core / ORM** – Uma solução de mapeamento de classes orienta a objetos para as tabelas do banco de dados que permite configurar rapidamente o banco e oferece mecanismos para fazer manipulações sobre os dados armazenados de maneira eficiente;
* **DTO** – Objetos que ajudam a fazer transferência de dados na rede e entre as camadas da aplicação, evitando o desperdício de informações em operações que não serão utilizadas;
* Compartilhado *(Alterdata.api.Shared)*

Camada onde reside o código que é compartilhado por todas as demais camadas e que oferece funcionalidades de caráter genérico para dar suporte ao sistema.

* Testes Automatizados *(Alterdata.api.IntegrationTest)*

Esta camada armazena um conjunto de testes automatizados de integração utilizados para cobrir alguns gaps dos testes unitários. É responsável por validar a interação entre os diferentes componentes da aplicação que pertencem a camadas diferentes e fazem uso de dependências externas. O objetivo é verificar se dados estão retornando corretamente ou se o comportamento desejado foi operado no sistema após a execução de alguma funcionalidade.

Como framework de testes foi selecionado o NUnit que é uma das 3 opções mais utilizadas na plataforma .NET para ajudar no controle de qualidade.

# Relação de Padrões de projetos e Bibliotecas

Abaixo segue uma relação das soluções que foram adotadas no projeto tendo em vista a sua facilidade de manutenção e reusabilidade e a influência na produtividade do time.

* **JWT** – Mecanismo de autenticação utilizado para garantir a segurança da aplicação e permitir o cadastro de usuários com atribuição de permissões;
* **Entity Framework Core** – ORM para fazer o mapeamento entre tabelas e classes orientadas a objetos tornando o desenvolvimento orientado a dados mais produtivo com a geração automática de tabelas no banco de dados e seus relacionamentos;
* **Microsoft Identity Membership** – Conjunto de classes que fornecem funcionalidade para cadastro, validação e autenticação de usuários com processo de geração de tabelas de permissões e perfis para controlar o acesso a aplicação;
* **Injeção de dependência** – Recurso que ajuda a diminuir o acoplamento entre as classes do sistema facilitando a sua manutenção e garantindo a testabilidade fornecendo as dependências utilizadas pelas classes de maneira externa;
* **Factory Method** – solução para centralizar o processo de criação de objetos em um único lugar;
* **Adapter** – mecanismo adotado para envolver código de bibliotecas de terceiros e garantir facilidade na troca de tecnologias caso seja necessário futuramente sem causar grande impacto no código existente;
* **Repository Pattern** – padrão usado para generalizar o acesso a dados através do mecanismo de herança e facilitar a implementação de operações de CRUD garantindo uma maior reutilização das funcionalidades implementadas.
* **Mapper** – Utilizado para fazer o mapeamento entre propriedades comuns de diferentes objetos.

# Banco de Dados

O banco de dados adotado na aplicação é o PostgreSQL 12.4. Foi utilizado o ORM EntityFrameworkCore para fazer a integração da aplicação com o banco, assim como a criação de todas as tabelas usando a abordagem Code First, onde o banco de dados é gerado a partir do modelo de classes da aplicação.

Para sincronizar as alterações ocorridas no modelo de dados e a estrutura física do banco está sendo utilizado o recurso Migrations disponível no ORM, tornando mais produtivo mudar colunas e tipos de dados quando necessário.

Para configurar o acesso ao banco de dados é necessário adicionar as credenciais do banco instalado no servidor na string de conexão que fica dentro do arquivo **appSettings.json**, informando o endereço do servidor, a porta do PostgreSQL (por padrão é a 5432), a senha e o banco dedados. Com essas informações devidamente fornecidas a aplicação se encarregará de fazer a conexão e gerar a estrutura inicial do banco com todas as classes de entidades devidamente mapeadas para as respectivas tabelas.

# COnsiderações Finais