**ALTERDATA SOFTWARE**

**SVR REST API**

**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](#_Toc48858923)

[2. Sobre o SVR 4](#_Toc48858924)

[3. A Web API 4](#_Toc48858925)

[4. Arquitetura 5](#_Toc48858926)

[5. Relação de Padrões de projetos, Frameworks e Bibliotecas Adotadas 7](#_Toc48858927)

[6. Banco de Dados 8](#_Toc48858928)

# 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 **SVR (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 às necessidades dos seus clientes.

# A 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 com armazenamento em um banco de dados PostgreSQL.

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

Para mais informações sobre como consumir a API e os formatos de dados suportados acesse o seguinte endereço quando o projeto estiver em execução: <https://localhost:44332/swagger/index.html>.

# Arquitetura

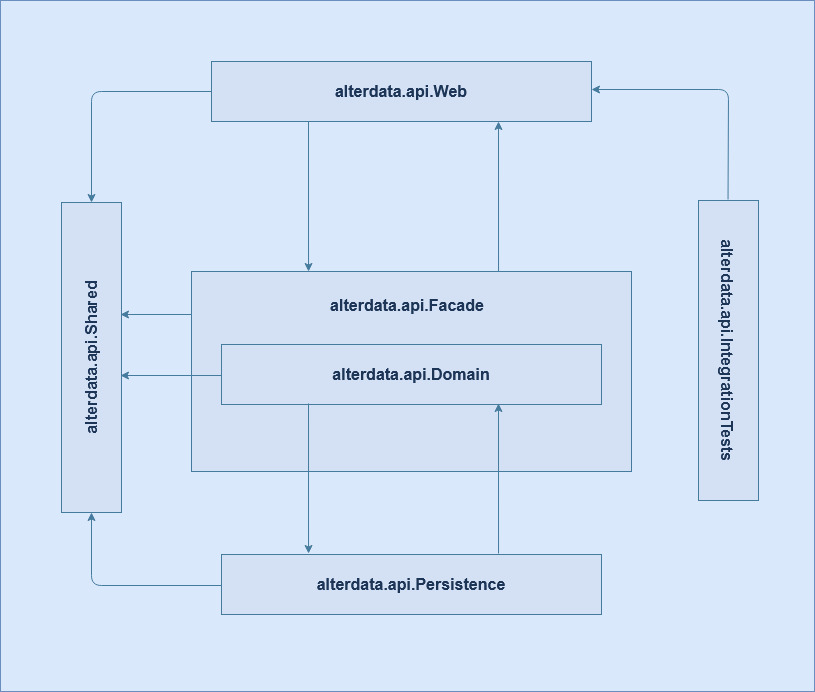


Fig. 1. Setas indicam o fluxo e a visibilidade de uma camada em relação à outra

Para este projeto foi selecionado uma arquitetura multicamadas (n-tier) onde cada uma possui uma responsabilidade e abstrai um importante aspecto do sistema. Esta divisão em camadas ajuda a separar os conceitos e oferece alguns benefícios para organizar o projeto facilitando a sua manutenção.

* Web *(Alterdata.api.Web)*

Esta camada hospeda os controles que ficam responsáveis por manipular as requisições HTTP que chegam por meio de uma aplicação externa, redireciona as solicitações para a camada debaixo, 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 no 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 reduz a quantidade de classes com as quais os controles precisam interagir, envolvendo a camada de negócios e ficando responsável por orquestrar operações que envolva a chamada de diferentes métodos na aplicação para executar uma operações específicas.

* Domínio *(Alterdata.api.Domain)*

A camada de domínio é onde reside a lógica de negócios da aplicação, ou seja, onde ficam as regras do sistema que representam sua principal razão de existir e agregam valor ao negócio do cliente (os requisitos funcionais). Também é onde reside recursos indispensáveis para a aplicação que faz o uso de 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 armazenadas as entidades que representam objetos do domínio do negócio e outras abstrações importantes 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.

* 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, como formatadores e classes de utilidades.

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

Esta camada armazena um conjunto de testes automatizados de integração utilizados para cobrir alguns gaps dos testes unitários em um projeto de Web API. É responsável por validar a interação entre os diferentes componentes da aplicação e que fazem uso de dependências externas. O objetivo é verificar se os dados estão retornando corretamente ou se os comandos executados no sistema causaram as alterações esperadas.

Como framework de testes foi selecionado o NUnit, que oferece atributos e funcionalidades para ajudar no controle de qualidade.

# Relação de Padrões de projetos, Frameworks e Bibliotecas Adotadas

* **JWT** – Mecanismo de autenticação utilizado para garantir a segurança da aplicação com a geração de um Token criptografado para permitir a identificação de usuários e suas permissões para controlar o acesso às funcionalidades disponibilizadas no servidor;
* **Entity Framework Core** – ORM utilizado para fazer o mapeamento de classes orientadas objetos para as tabelas do banco de dados PostgreSQL, tornando o desenvolvimento orientado a dados mais produtivo com a geração automática de tabelas e relacionamentos;
* **Microsoft Identity Membership** – Uma plataforma que oferece funcionalidades de autenticação, autorização e registro de usuários fornecida pela Microsoft para ajudar a garantir a segurança da aplicação e abstrair a criação de tabelas de perfis e permissões;
* **Injeção de dependência** – Recurso que ajuda a diminuir o acoplamento entre as classes do sistema facilitando a sua manutenção e 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 e fornecer uma alternativa ao recurso de Injeção de dependência, uma vez que permite adicionar configurações às classes que serão instanciadas;
* **Adapter** – mecanismo adotado para envolver código de bibliotecas de terceiros e garantir facilidade no uso e na troca futura de tecnologias utilizadas caso sem causar um grande impacto no código existente;
* **Repository Pattern** – Padrão de projetos 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. Também garante um isolamento entre as camadas de Domínio e Acesso a Dados facilitando sua manutenção.
* **Mapper** – Utilizado para fazer o mapeamento entre propriedades comuns de diferentes objetos para garantir que somente os dados requisitados sejam transportados entre as camadas e retornados para as aplicações clientes, evitando o desperdício de informações e a propagação de objetos com propriedades nulas ou que não serão utilizadas.

# Banco de Dados

O banco de dados adotado na aplicação é o PostgreSQL 12.4. Foi utilizado o ORM EntityFrameworkCore em conjunto 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. Os relacionamentos foram criados através da composição entre os objetos com propriedades de navegação inseridas em cada classe que representam as chaves primária e estrangeiras.

Para sincronizar as alterações ocorridas entre o modelo de dados e a estrutura física do banco está sendo utilizado o recurso *Migrations* disponível no ORM, acelerando o processo de alteração de 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 de dados. Com essas informações devidamente fornecidas a aplicação se encarregará de fazer a conexão com o SGBD e gerar a estrutura inicial do banco com todas as classes de entidades devidamente mapeadas para as respectivas tabelas usadas pela aplicação.