MeuScrum

Descrição de Arquitetura de Software

1. Propósito

Este documento descreve a filosofia, decisões, restrições, justificativas e mecanismos arquiteturais do sistema de software para **gestão de projetos ágeis com base no framework Scrum**, permitindo o gerenciamento de histórias de usuário, backlogs e planos de sprint, por meio de uma aplicação web acessível a partir de qualquer estação de trabalho disponibilizada de forma Web.

O grande propósito deste documento é comunicar aos principais stakeholders como o sistema está organizado sem precisar ler cada linha de código. Ela também ajuda a capturar por que certas decisões arquiteturais foram tomadas, como por exemplo, porque foi adotado uma arquitetura de aplicação web, e quais as vantagens dessa arquitetura para esse modelo de projeto.

Este documento também serve de guia para desenvolvedores implementarem de acordo com o plano arquitetural, evitando que cada módulo “faça do seu jeito” e que o sistema vire um emaranhado. Isso facilita o processo de manutenção futura do software e eventuais evoluções do mesmo. Também, o documento ajuda a evidenciar como os atributos de qualidade são tratados e definir padrões de desenvolvimento.

2. Filosofia e Metas Arquiteturais

O sistema é projetado para oferecer uma **plataforma colaborativa**, **intuitiva** e **robusta**, que suporte múltiplos projetos simultâneos, cada um com papéis e permissões específicas. O foco está na **clareza e separação de responsabilidades** para facilitar o processo de manutenção e, simultâneamente, prover um sistema simples e fácil uso ao usuário. Para isso, foi optado pelo uso de uma arquitetura em camadas, que é o modelo mais utilizado em aplicações web.
A arquitetura baseada em camadas de menor complexidade, como é o caso desse projeto, costumam ser divididas em três camadas, onde, em alguns casos, as camadas de negócios e persistência atuam como uma camada. A ideia principal desse modelo é separar o sistema em **camadas lógicas**, cada uma com responsabilidades bem definidas.

Metas arquiteturais:

- **Separação de Responsabilidades**, cada camada deve ter responsabilidades específicas, e não interferir umas nas outras, permitindo uma redução do acoplamento.
- **Encapsulamento de Lógica**, cada camada deve encapsular sua própria lógica, expondo apenas o necessário à camada superior.
- **Redução de Acoplamento e Alta Coesão**, as dependências devem ser verticais e controladas.
- **Reutilização e Consistência**, cada camada pode ser reutilizada em outros contextos, mantendo comportamentos previsíveis.
- **Facilidade de Manutenção e Evolução**, alterações em uma camada devem impactar minimamente as demais.
- **Testabilidade**, cada camada pode ser testada isoladamente, simulando dependências.
- **Escalabilidade e Desempenho Controlado**, separar as camadas permite escalar partes específicas conforme a demanda.
- **Clareza Arquitetural e Documentação**, o sistema deve ter uma estrutura compreensível e documentada.
- **Segurança e Controle de Acesso por Camada**, cada camada pode implementar regras específicas de segurança.
- **Facilidade de Integração e Substituição Tecnológica**, a divisão em camadas permite substituir tecnologias sem comprometer o todo.

3. Premissas e Dependências

Levando em conta a natureza do projeto, que se trata de um site de gestão de múltiplos projetos com definição de papéis para os membros da equipe, e que tem a finalidade especial de adotar a metodologia Scrum, foram levantadas algumas premissas com o intuito de sustentar o desenvolvimento desta aplicação.
Tendo como base dessas premissas o fato de que o sistema deve atender as necessidades de equipes multifuncionais (pensando especialmente nos rótulos desenvolvedor, product owner, scrum master) distribuídas e trabalhando de forma interativa, foi verificada uma necessidade incisiva de que a Arquitetura de aplicação web sustente uma colaboração em tempo real, controle de acesso e segurança de artefatos, disponibilidade constante da plataforma, escalabilidade para múltiplos times e projetos e facilidade de evolução conforme mudanças metodológicas ou de ferramenta.
Por esses fatores, foram levantadas as seguintes **premissas**:

- A **arquitetura** deve ser estruturada em **camadas**: Deve garantir separação de responsabilidades, manutenção e escalabilidade através da filosofia de arquitetura em camadas.
- Uso de arquitetura orientada a serviço ou micro serviço: Pensando no aspecto de implementação, a aplicação deve permitir distribuir funcionalidades em serviços independentes, facilitando deploy e evolução contínua.
- A camada de domínio deve ser independente de frameworks e bancos: Ou seja, a aplicação deve seguir o princípio de **Hexagonal** ou Clean Architecture, permitindo trocar **banco** ou **linguagens de implementação** sem reescrever a lógica de negócio.
- As interações entre camadas/serviços devem usar **APIs RESTful** com **HTTPS e JWT/OAuth2:** Garante interoperabilidade e segurança.
- Interface web responsiva e adaptável: Usuários acessam via desktop, tablet ou celular
- Implantação em ambiente **cloud-native**: Facilita escalabilidade horizontal e tolerância a falhas.
- Uso de **WebSockets** ou **event-driven architecture:** Permite atualização instantânea de quadros Kanban, tarefas, notificações e sprints.
- Todas as operações críticas devem ser logadas: Essencial para rastrear decisões, alterações de backlog e entregas.
- Métricas e logs centralizados: Deve garantir suporte à manutenção e diagnóstico de falhas.
- Estrutura do sistema deve refletir **artefatos e eventos do Scrum:** Deve possibilitar a realização de Product Backlog, Sprint Backlog, Burndown Chart, Retrospectiva e Sprint Review.

A partir deste levantamento, foram observadas algumas **dependências** que se veem fundamentais para o desenvolvimento do projeto:

- **Serviços de nuvem (AWS, Azure, GCP)**: Hospedagem, banco de dados, balanceamento de carga e armazenamento de arquivos
- **Banco relacional (PostgreSQL) e NoSQL (MongoDB):** Para lidar com dados estruturados e não estruturados
- **Integrações Externas (Github, Gitlab, etc):** Aumenta o valor da aplicação ao integrar-se com o ecossistema de desenvolvimento.
- **Serviços de identidade:** Simplifica gestão de acesso e Single Sign-On (SSO)
- **Conhecimento em Scrum, DevOps, APIs REST, CI/CD:** A qualidade da arquitetura depende da maturidade técnica da equipe.
- **Compliance com LGPD e boas práticas OWASP:** Necessário para proteção de dados e prevenção de vulnerabilidades.
- **Estimativa de tempo de resposta inferior a 2 segundos e disponibilidade 99,9%**: Estabelecem metas claras de desempenho e confiabilidade.

4. Requisitos Arquiteturalmente Significativos

Abordando agora os requisitos arquiteturais mais significativos para esse projeto, ou seja, os requisitos que afetam as principais decisões estruturais da arquitetura, pode-se descrever:

No quesito de **requisitos não-funcionais:**

Categoria	Referência
Disponibilidade O sistema deve estar disponível 24/7, com tempo de inatividade máximo de 0,1% (SLA 99,9%). — Implica uso de deploy redundante, balanceamento de carga e replicação de dados.	Microsoft Patterns – Availability
Desempenho e Escalabilidade O sistema deve suportar até 10.000 usuários simultâneos com tempo de resposta médio inferior a 2 segundos. — Implica caching, CDN, micro serviços escaláveis e otimização de consultas.	Microsoft – Performance and Scalability Guide
Segurança A aplicação deve proteger dados de projeto e usuários com autenticação OAuth2/JWT, criptografia HTTPS/TLS, e aderência à LGPD.	OWASP Web Security Guide, Microsoft Security Guidance
Integridade e Consistência dos Dados Operações simultâneas (como atualização de backlog) devem ser atomicamente consistentes. — Implica uso de transações distribuídas ou eventual consistency em sistemas distribuídos.	SEI/CMU Software Architecture Guide
Manutenibilidade e Evolução O código deve ser modular e permitir substituição de componentes sem afetar o sistema (ex.: trocar banco ou front-end). — Adotar arquitetura em camadas e Clean/Hexagonal Architecture.	Microsoft Layered Application Guide
Usabilidade A interface deve seguir princípios de UX ágil, com interação fluida, responsiva e intuitiva. — Baseada em frameworks modernos (React, Angular, Vue).	Microsoft – Web UI Development Guide
Portabilidade e Implantação O sistema deve poder ser implantado em ambientes cloud (AWS, Azure, GCP) via containers (Docker, Kubernetes).	Microsoft Cloud Architecture Guide

Já os **requisitos funcionais de alto nível**, são:

Categoria	Referência
Gestão de Projetos e Tarefas O sistema deve permitir criar, priorizar e acompanhar backlogs, sprints e tarefas.	Scrum Guide 2020
Gestão de Usuários e Papéis O sistema deve permitir controle de papéis e de acesso por papéis Scrum (Product Owner, Scrum Master, Desenvolvedor).	Scrum.org – Roles
Colaboração em Tempo Real Atualizações em tarefas e quadros devem refletir em tempo real para todos os membros.	Simform – Web Application Architecture
Gestão de Artefatos e Documentos Permitir upload, versionamento e acesso controlado de documentos e evidências de projeto.	Microsoft Patterns – Data Access Layer
Integrações Externas Integração com GitHub ou GitLab via APIs REST.	Microsoft – Integration and Messaging Patterns

5. Decisões, Restrições e Justificativas

Decisão	Justificativa	Recomendação	Restrições
O sistema será organizado em camadas (Apresentação, aplicação, domínio e dados	Promover separação de responsabilidade, facilita a manutenção, teste e evolução tecnológica e permitir substituir tecnologias sem afetar outras partes do sistema	Cada camada deve expor interfaces bem definidas e a comunicação entre camadas deve ser somente descendente.	Não devem haver saltos de camadas
A comunicação entre a camada de apresentação e aplicação será feita via REST API sobre HTTP/HTTPS com formato JSON.	Facilitar a integração com outros clientes e permitir autenticação padronizada.	Utilizar verbos HTTP semânticos e a resposta deve seguir o padrão HTTP status code	Evitar payloads excessivos, não expor endpoints sem autenticação e fora do padrão REST e não incluir lógica de negócio na API Controller

A camada de domínio implantou o modelo de negócio orientado a objetos, com entidades, agregados e serviços de domínio.	Serve para centralizar a lógica de negócio em um local único e reutilizável, evitar duplicação de código e suportar testes de unidade independente da infraestrutura.	Criar classes de domínio imutáveis sempre que possível, usar Value Object para representar conceitos variáveis e validar regras de negócio no domínio.	Não misturar código persistente dentro do domínio e evitar dependência direta de frameworks no domínio
Implementar autenticação e autorização via OAuth2 + JWT, com controle de papéis (roles) por tipo de usuário Scrum.	Com o intuito de proteger recursos sensíveis, garantir confidencialidade e conformidade com LGPD e simplificar integração com provedores externos.	Toda API deve exigir token JWT válido e os papéis e permissões devem ser definidos em nível de endpoint e serviço	Não deve haver armazenamento de senhas em texto puro, nem incluir tokens em URLs e deve-se evitar validação de autenticação apenas no Front-end.
A arquitetura deve suportar carga crescente e balanceamento horizontal.	O sistema será multiusuário e potencialmente acessado globalmente, desta forma, deve-se evitar gargalos em um único ponto de falha.	Implementar cache de dados e aplicar monitoramento de desempenho.	Não deve haver realização de operações pesadas em tempo real na camada de aplicação e deve-se evitar dependências síncronas entre micro serviços críticos.
Implementar WebSockets ou SignalR para atualizações de sprint, backlog e quadro Scrum em tempo real.	Trata-se de um requisito funcional essencial para sincronização de tarefas e equipes distribuídas e o que ocasionará aumento da percepção de “agilidade” e engajamento do usuário.	Orienta-se o uso de canais dedicados para eventos de atualização, garantia de controle de sessão e permissões para eventos e registro de logs de eventos para auditoria.	Não deve haver tráfego de dados confidenciais em eventos WebSockets e deve-se evitar atualizações em broadcast desnecessariamente.
Toda camada deve ser testável isoladamente e de forma automatizada.	Reduz a regressão e garante confiabilidade em releases ágeis e incrementais.	Adoção de frameworks de testes, utilização de mocks para dependências externas e definição de cobertura mínima de 80%	Não fazer deploy sem execução dos testes automatizados e evitar dependência de ambiente externo em testes unitários.

6. Mecanismos Arquiteturais

Mecanismo de separação em camadas

Deve estruturar o sistema em quatro camadas lógicas: Apresentação (UI), Aplicação (Serviços/API), Domínio (Regras de Negócio) e Dados (Persistência). Cada camada possui responsabilidade própria e comunicação controlada. Esse mecanismo visa garantir baixo acoplamento, clareza estrutural e facilidade de manutenção.

Injeção de Dependência

Visa gerenciar as dependências entre camadas e componentes de forma automatizada. Esse mecanismo visa facilitar a testabilidade, substituição de implementações e evitar acoplamento rígido entre camadas.

Mecanismo de Comunicação RESTful

Implementa comunicação entre cliente e servidor via API RESTful com JSON sobre HTTP/HTTPS, seguindo princípios de *statelessness* e versionamento de API. Esse mecanismo permite integração padronizada, compatível com aplicações web, mobile e serviços externos.

Mecanismo de Autenticação e Autorização

Implementa segurança baseada em OAuth2 + JWT. Cada usuário possui *token* com permissões específicas. Isso visa garantir segurança, controle de acesso e conformidade com LGPD, além de permitir integração com provedores externos.

Mecanismo de Persistência e Acesso a Dados

A camada de dados usa padrão Repositóry e ORM para abstrair o banco de dados relacional. Isso garante a abstração da fonte de dados, independência de tecnologia, e facilidade de manutenção do modelo de persistência.

Mecanismo de Modelagem de Domínio

O núcleo do sistema segue princípios de Domain-Driven Design, com entidades, agregados, value objects e serviços de domínio. Esse mecanismo centraliza e organiza as regras de negócio, reduzindo duplicações e facilitando testes unitários.

Mecanismo de Mapeamento de Dados

Transforma objetos entre diferentes camadas, garantindo isolamento entre camadas e evitando vazamento de modelos internos para a API pública.

Mecanismo de Transação e Consistência

Gerencia transações ACID entre múltiplas operações no banco. Isso assegura integridade dos dados em operações críticas.

Mecanismo de Cache e Desempenho

Implementar cache de resultados e sessões para melhorar o tempo de resposta, reduzir carga no banco e aumentar a escalabilidade do sistema.

Mecanismo de Comunicação em Tempo Real

Utilizar WebSockets para notificações e atualização instantânea de dados. Isso permitirá a colaboração em tempo real entre membros da equipe. Essencial para experiência ágil e responsiva.

Mecanismo de Testes e Mocking

Define infraestrutura para testes unitários, de integração e de aceitação. Inclui *mocks*, *stubs* e *test doubles*. Isso garante confiabilidade, qualidade contínua e segurança nas mudanças.

Mecanismo de Deploy e Entrega Contínua

Automatização de build, testes e deploy via pipelines. Favorece entregas ágeis, rastreabilidade e redução de erros manuais.

Mecanismo de Monitoramento e Logging

Centralizar logs e métricas, incluindo alertas e rastreamento de requisições. Isso viabiliza observabilidade,

auditoria e resolução rápida de falhas.

Mecanismo de Versionamento e Migração de Banco de Dados

Controla a evolução do schema via ferramentas de migração, evitando inconsistências de dados e facilidade no controle de versão do modelo relacional.

Mecanismo de Configuração Centralizada

Armazena parâmetros de ambiente de forma segura, permitindo separação de configuração e código, além de segurança e flexibilidade de deploy.

Mecanismo de Internacionalização e Localização

Adapta o conteúdo textual e formatação para diferentes idiomas e regiões. Isso facilita a adoção global e melhora a experiência do usuário.

Mecanismo de Documentação Automática de API

Geração de documentação interativa das APIs, facilitando a comunicação com desenvolvedores e integrações externas.

7. Abstrações Principais

As principais abstrações desse projeto derivam diretamente dos conceitos fundamentais do Scrum e das necessidades de colaboração, rastreabilidade e controle de artefatos. Dentre essas abstrações podemos citar:

- **Projeto**: Representa o escopo geral de um produto ou sistema em desenvolvimento. É o container principal que agrupa sprints, backlog, artefatos e equipe.
- **Equipe:** Conjunto de membros que atuam no projeto. Cada membro tem um papel Scrum.
- **Usuário**: entidade que representa qualquer membro autenticado do sistema.
- **Papel (Role)**: Define o tipo de atuação dentro da metodologia Scrum (PO, SM, Developer, Admin)
- **Backlog de Produto**: Lista ordenada de requisitos, histórias de usuário e tarefas do produto.
- **Sprint:** Ciclo de tempo fixo onde um conjunto de itens de backlog é planejado, executado e revisado.
- **Tarefas:** Subdivisão de um item de backlog, com atribuição direta a um membro da equipe.
- **Artefato:** Documentos e evidências de projeto.
- **Reunião:** Representa eventos Scrum (Daily, Review, Retrospective, Planning).
- **Métrica:** Indicadores de qualidade.
- **Notificação:** Comunicação assíncrona entre membros sobre eventos.
- **Autenticação:** Responsável por autenticar usuários via OAuth2/JWT.
- **Autorização:** Controlar permissões com base em papéis.
- **Repositório:** Interface para acesso e persistência de dados de domínio.
- **Serviço de Domínio:** Implementa regras de negócio complexas que envolvem múltiplas entidades.
- **Serviço de Aplicação:** Coordena fluxos de caso de uso.
- **Mapper / Data Transfer Object:** Faz a conversão entre entidades de domínio e objetos de transporte da API.
- **Gerenciador de Transações:** Garante atomicidade e consistência de operações.
- **Serviço de Armazenamento:** Gerência upload e versionamento de arquivos.
- **Serviço de Relatórios:** Gera relatórios e métricas de projeto.
- **Padrão em Camadas:** Estrutura lógica com separação entre UI, Application, Domain e Data.

8. Camadas ou Framework Arquitetural

A **Arquitetura em Camada** é um dos padrões mais difundidos em sistemas corporativos, e é especialmente adequada para aplicações web que exigem separação de responsabilidades, facilidade de manutenção, testabilidade e clareza estrutural.

Neste projeto, este padrão foi adotado como estrutura central da solução, servindo de base para a organização do código-fonte, das responsabilidades, do fluxo de comunicação entre módulos e das decisões tecnológicas. O intuito desse padrão é separar a aplicação em blocos lógicos bem definidos, cada um com responsabilidades específicas, reduzir acoplamento entre partes do sistema e organizar o sistema de modo que cada camada trate de um tipo específico de preocupação.

Normalmente essa arquitetura está dividida em quatro camadas: Camada de Apresentação, Camada de Aplicação, Camada de Domínio e Camada de Infraestrutura / Dados.

A **Camada de Apresentação**, é responsável por fornecer a interface visual, receber e enviar dados ao usuário final, traduzir eventos da interface em requisições à API, exibir dados de domínio recebidos da camada de aplicação, validar dados de entrada a nível de interface e gerenciar sessões, interações e notificações em tempo real. Normalmente ela é composta por componentes UI, normalmente feitos em React ou Angular, Layouts, formulários, widgets, gráficos, Controladores REST, que vão integrar ao back-end, views e models de UI e WebSocket client para atualizações de sprint e tarefas. É importante ressaltar que essa camada não contém lógica de negócio e traduz as intenções do usuário para operações de aplicação.

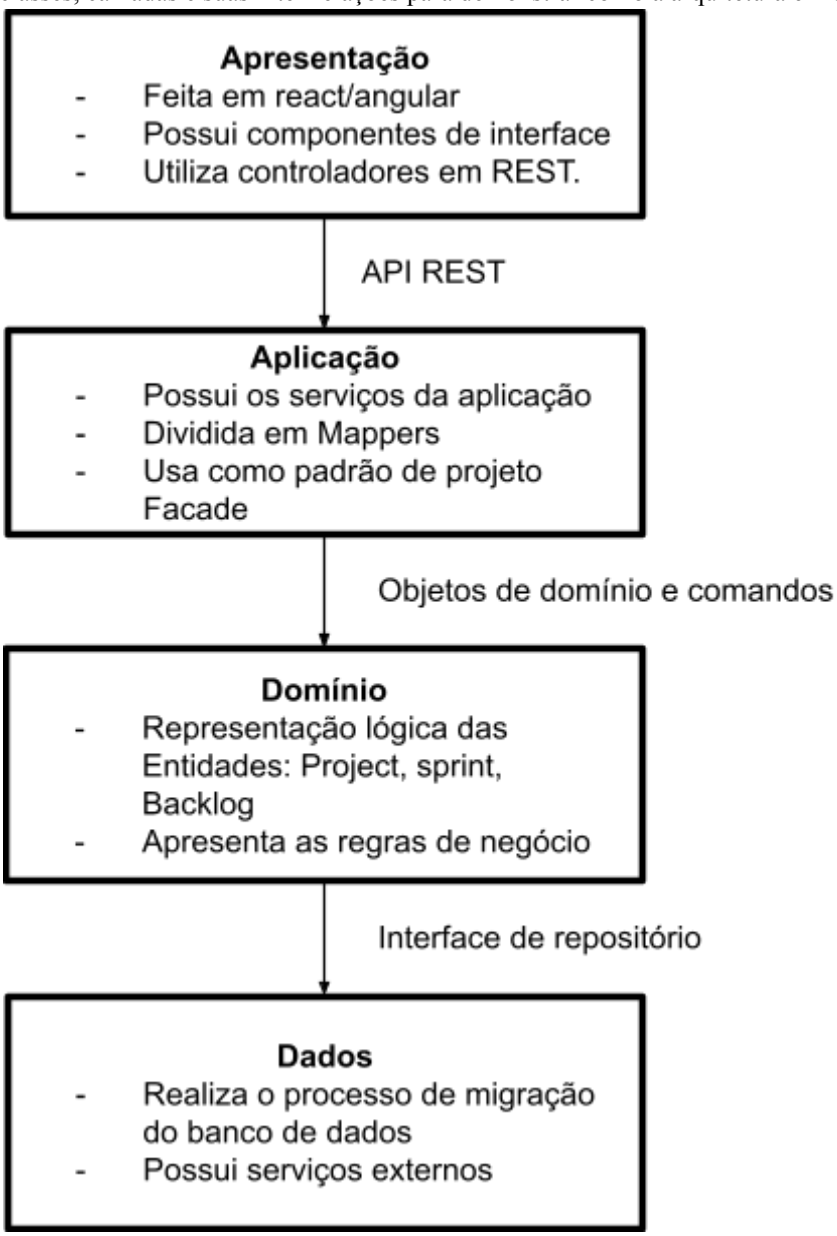
A **Camada de Aplicação**, é responsável por coordenar casos de uso e fluxos de aplicação, orquestrar interações entre UI e domínio, aplicar regras de autorização, controlar transações de processo, validar dados de forma contextual e expor serviços para consumo externo. Usualmente é composta por serviços de aplicação, Facades que agrupam casos de uso, Mappers e Validadores de Entrada e essa camada, tal qual a anterior, não possui lógica de domínio e é responsável por transações e fluxos.

A **Camada de Domínio**, é considerada o coração do sistema, ela é responsável por conter as regras de negócio fundamentais, representar o modelo de domínio Scrum, definir entidades, agregados, value objects, garantir invariantes, consistência e validade do domínio e armazenar lógica crítica. Usualmente composta por entidades, value objects, domain services e regras, políticas e restrições de domínio. Essa camada é totalmente independente das outras camadas e não tem conhecimento de tecnologias das demais camadas, como HTTP ou Banco de Dados, permitindo manter uma pureza lógica e garantindo testabilidade isolada..

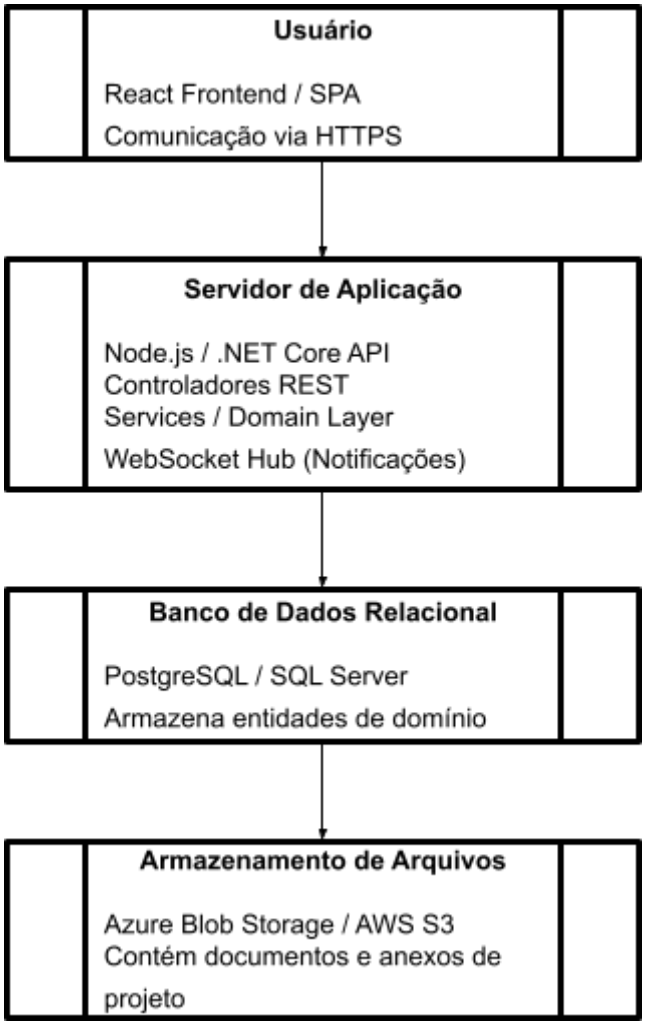
A **Camada de Infraestrutura / Dados**, é a camada responsável por fornecer implementação concreta de persistência de dados, interagir com banco de dados relacional, expor repositórios concretos e fornecer integrações externas. Essa camada é tipicamente composta por repositórios, configurações, adaptadores e interfaces técnicas, unidades de trabalho e tabelas de bancos de dados e schemas e seus princípios consistem em implementar abstrações definidas no domínio e concentrar detalhes técnicos que podem ser substituídos.

9. Visualização Arquitetural

Visão lógica: Tem por objetivo representar a estrutura lógica e de software do sistema, incluindo módulos, pacotes, classes, camadas e suas inter-relações para demonstrar como a arquitetura em camadas organizará a aplicação.



Visão operacional: Tem o intuito de descrever como o sistema se comporta em tempo de execução, mostrando nós físicos, componentes implantados e processos ou threads que interagem entre si.



Front End: Interface visual e controle de interface do usuário
API Gateway / Backend: Gerência autenticação, autorização e execução de casos de uso.
WebSocket Server: Gerencia notificações e eventos em tempo real.
Banco de Dados: Persistência relacional e controle transacional.
Storage Service: Armazenamento de artefatos de projeto.
CI/CD Pipeline: Automatiza build, testes e deploy.

Visão de Casos de Uso: Quanto às visões de caso de uso, essa destaca os casos de uso mais relevantes para a arquitetura, ou seja, aqueles que impactam mecanismos estruturais, segurança, persistência ou comunicação. Para uma compreensão detalhada desta visão neste projeto, recomenda-se a leitura do documento Projeto de interface com o usuário. Nele, é detalhado cada um dos casos de uso do projeto e apresentado em wireframes os casos de uso mais relevantes para este projeto.