SERVIÇO PÚBLICO FEDERAL
MJSP
COORDENAÇÃO DE SOLUÇÕES DE TECNOLOGIA DA INFORMAÇÃO - CST/CGTEC/STI/SE/MJSP

PADRÕES ARQUITETURAIS E TECNOLÓGICOS

ANEXO IX DO TERMO DE REFERÊNCIA

Processo Administrativo nº 08006.000703/2025-56

Arquitetura de Referência para Soluções em T.I. — elaborado pela CGTEC/CAQ (Coordenação-Geral de Soluções de Tecnologia da Informação / Coordenação de Arquitetura, Qualidade, Testes e Automação)

Histórico de Revisões

Data	Versão	Descrição	Autor
09/03/2026	1.0	Documento original "Arquitetura de Referência", elaborado pela CGTEC/CAQ (Coordenação-Geral de Soluções de Tecnologia da Informação / Coordenação de Arquitetura, Qualidade, Testes e Automação)	Guilherme Borges
21/08/2026	1.1	Formalização como Anexo IX do Termo de Referência nº 11/2026 (processo 08006.000703/2025-56), instrumento de governança do processo de desenvolvimento de software previsto no item 4.6.1, alínea "d", do Anexo I da Portaria SGD/MGI nº 750, de 20 de março de 2023, incluído pela Portaria SGD/MGI nº 4.777, de 16 de junho de 2026 — em atendimento ao item 19.16 do Parecer SIRT nº 3051/2026/MGI; formatação no padrão dos anexos do Termo de Referência (capa, brasão, histórico de revisões, cabeçalho corrido), sem alteração do conteúdo técnico original. PENDENTE de validação formal pela CGISE/STI antes de vinculação definitiva ao TR.	Guilherme Borges
21/08/2026	1.2	Inclusão de PostgreSQL e SQL Server (bancos relacionais alternativos ao Oracle Database) e Databricks (Big Data/Analytics) no item 5.1.3 (Persistência e Dados) e no glossário do item 1.1, ausentes do documento original da CGTEC/CAQ. Único ponto do Anexo com conteúdo técnico	Guilherme Borges

		acrescentado por esta formalização — demais itens são transcrição integral do original.	

1. Visão Geral



1.1. Definições

Spring Boot: Framework do ecossistema Spring utilizado para o desenvolvimento de aplicações Java baseadas no paradigma de inversão de controle (IoC) e injeção de dependências. O Spring Boot simplifica a configuração e inicialização de aplicações, promovendo padronização, modularidade e produtividade no desenvolvimento de serviços e aplicações corporativas.

Microserviços: estilo arquitetural no qual uma aplicação é estruturada como um conjunto de serviços pequenos, autônomos e fracamente acoplados, cada um responsável por uma capacidade de negócio específica. Esses serviços comunicam-se por meio de interfaces bem definidas, normalmente APIs, permitindo evolução independente, escalabilidade e maior resiliência do sistema.

Padrão MVC: Padrão arquitetural que separa a aplicação em três componentes principais: Model, responsável pela representação dos dados e regras de negócio; View, responsável pela apresentação das informações ao usuário; e Controller, responsável por receber as requisições, processar o fluxo da aplicação e coordenar a interação entre Model e View.

React: Biblioteca JavaScript utilizada para construção de interfaces de usuário em aplicações web, baseada no conceito de componentes reutilizáveis e atualização eficiente da interface por meio de renderização reativa. O React é amplamente utilizado no desenvolvimento de aplicações modernas de página única (SPA), promovendo modularidade, reutilização de código e melhor manutenção da camada de apresentação.

TypeScript: Superset do JavaScript que adiciona tipagem estática e recursos adicionais de linguagem, promovendo maior robustez e manutenção do código.

Vite: Ferramenta de build e empacotamento utilizada no desenvolvimento de aplicações web modernas. O Vite oferece inicialização rápida do ambiente de desenvolvimento e atualização instantânea de

módulos, proporcionando maior produtividade e desempenho no desenvolvimento de aplicações frontend baseadas em frameworks como React.

Material: Sistema de design desenvolvido pelo Google que estabelece diretrizes de interface, componentes visuais e padrões de interação para aplicações digitais, promovendo consistência visual, acessibilidade e melhor experiência do usuário.

SonarQube: Plataforma de análise contínua de qualidade de código utilizada para identificar vulnerabilidades, falhas de segurança e problemas de manutenção, contribuindo para a melhoria da qualidade e confiabilidade do software.

Robot Framework: Framework de automação de testes utilizado para validação funcional e testes de aceitação de sistemas. Permite a criação de testes automatizados de forma estruturada e reutilizável.

Test Automation: Estratégia de utilização de ferramentas e scripts automatizados para execução de testes funcionais e não funcionais, com o objetivo de aumentar a confiabilidade, repetibilidade e eficiência no processo de validação de software.

DevOps: Conjunto de práticas que integra desenvolvimento de software e operações de infraestrutura, promovendo automação, colaboração entre equipes e entrega contínua de aplicações.

CI/CD Pipelines: Processo automatizado de integração contínua e entrega contínua utilizado para compilar, testar e implantar aplicações de forma automatizada, garantindo maior velocidade e confiabilidade no ciclo de desenvolvimento.

Docker: Plataforma de containerização utilizada para empacotar aplicações e suas dependências em contêineres portáveis, garantindo consistência entre ambientes de desenvolvimento, teste e produção.

Kubernetes: Plataforma de orquestração de contêineres utilizada para gerenciamento, escalabilidade e automação da implantação de aplicações containerizadas em ambientes distribuídos.

Microsoft Azure: Plataforma de computação em nuvem utilizada para provisionamento de infraestrutura, serviços e recursos necessários à execução e escalabilidade de aplicações corporativas.

Spring Security: Framework de segurança do ecossistema Spring utilizado para implementação de autenticação, autorização e proteção de aplicações Java, permitindo integração com diversos mecanismos de identidade.

Keycloak (IAM): Solução open source de Identity and Access Management (IAM) utilizada para prover autenticação, autorização e gerenciamento de identidades nas aplicações institucionais. Permite implementar Single Sign-On (SSO), integração com protocolos padrão de mercado (OAuth2, OpenID Connect e SAML) e centralização das políticas de acesso. No modelo de referência, o Keycloak é a solução obrigatória, sendo o componente padrão para gerenciamento de identidade e controle de acesso nos sistemas.

RHBK® – Red Hat Build of Keycloak: Distribuição empresarial do Keycloak mantida pela Red Hat, que oferece suporte corporativo, atualizações certificadas e integração com o ecossistema Red Hat.

GovBr: Plataforma de identidade digital do Governo Federal utilizada para autenticação unificada de usuários em serviços públicos digitais, permitindo integração segura entre aplicações governamentais. É essencial integrar com Keycloak.

Oracle® Database: Sistema gerenciador de banco de dados relacional utilizado para armazenamento, gerenciamento e consulta de dados estruturados em aplicações corporativas.

PostgreSQL: Sistema gerenciador de banco de dados relacional open source, admitido como alternativa ao Oracle Database para aplicações corporativas e para ambientes em que a licença Oracle não se justifique, mantendo os mesmos requisitos de modelagem relacional e integridade transacional.

SQL Server: Sistema gerenciador de banco de dados relacional da Microsoft, admitido como alternativa ao Oracle Database em soluções cujo ambiente de hospedagem ou integração (ex.: Azure SQL) recomende essa escolha.

Databricks: Plataforma de engenharia e análise de dados em larga escala, baseada em Apache Spark, utilizada para processamento de cargas analíticas, Big Data e pipelines de dados que excedam a capacidade adequada de um banco relacional tradicional.

Apache Maven: Ferramenta de automação de build utilizada em projetos Java para gerenciamento de dependências, compilação, empacotamento e padronização do ciclo de vida de aplicações.

Sonatype Nexus Repository: Repositório de artefatos utilizado para armazenamento e distribuição de dependências, bibliotecas e pacotes de software utilizados no processo de build e integração contínua.

Github: Plataforma de hospedagem de código-fonte baseada em Git utilizada para controle de versão, colaboração entre desenvolvedores e automação de processos de desenvolvimento.

2. Finalidade

A Arquitetura de Referência constitui um conjunto de artefatos que integram a base de ativos reutilizáveis de uma organização. A finalidade desses artefatos é fornecer um ponto de partida estruturado para o desenvolvimento de arquiteturas de sistemas, promovendo padronização, consistência e reutilização de soluções tecnológicas.

Esses artefatos podem abranger desde padrões arquiteturais, mecanismos de arquitetura e frameworks, até soluções mais completas, com características conhecidas e comprovadas em uso. Podem ser aplicados de forma geral a diversos tipos de sistemas ou direcionados a classes específicas de aplicações e domínios de negócio.

A adoção de arquiteturas de referência consolidadas é uma estratégia eficaz para lidar com requisitos não funcionais, especialmente aqueles relacionados à qualidade do software, como desempenho, segurança, escalabilidade, disponibilidade e manutenibilidade. Ao utilizar arquiteturas previamente testadas e validadas em cenários reais, aumenta-se a previsibilidade e a confiabilidade das soluções desenvolvidas.

As arquiteturas de referência podem ser descritas em diferentes níveis de abstração e sob diversos pontos de vista arquiteturais. Esses pontos de vista são tradicionalmente organizados conforme o modelo 4+1 de visões arquiteturais, permitindo representar a arquitetura sob diferentes perspectivas.

Dessa forma, o arquiteto de software pode selecionar a visão mais adequada ao contexto, seja para representar apenas o design arquitetural, seja para descrever design e implementação, em diferentes níveis de detalhamento e maturidade.

2.1. Objetivo

O objetivo deste documento é apresentar e formalizar o modelo arquitetural adotado para o desenvolvimento dos sistemas do Ministério da Justiça, estabelecendo diretrizes técnicas e padrões a serem observados pelas equipes de desenvolvimento.

Neste documento são descritos os princípios arquiteturais, as diretrizes técnicas e a organização das camadas que compõem o modelo de arquitetura adotado, bem como aspectos relacionados a segurança, desempenho, integração e qualidade das soluções de software.

Adicionalmente, este documento tem como finalidade orientar os profissionais técnicos envolvidos no desenvolvimento e manutenção dos sistemas, fornecendo diretrizes claras sobre tecnologias, padrões de desenvolvimento, boas práticas e mecanismos arquiteturais recomendados para os projetos institucionais.

2.2. Metas e Restrições

A meta da arquitetura é suportar os requisitos funcionais (de negócio) e os requisitos não funcionais da aplicação, tais como: manutenibilidade, interoperabilidade, portabilidade, reusabilidade, confiabilidade, integridade e segurança.

A arquitetura definida é baseada na plataforma Java, utilizando o framework Spring Boot para o desenvolvimento de aplicações corporativas. Essa abordagem permite a construção de serviços e componentes de negócio desacoplados, baseados em injeção de dependência, inversão de controle (IoC) e organização em camadas bem definidas, favorecendo a padronização e a produtividade no desenvolvimento.

Por meio dessa arquitetura, torna-se possível desenvolver componentes de negócio, serviços e APIs reutilizáveis, promovendo maior modularidade e facilitando a evolução e manutenção dos sistemas ao longo do tempo.

Com a arquitetura de referência definida neste documento, adota-se a estratégia de separação entre camadas server-side e client-side, onde os componentes de backend responsáveis pela lógica de negócio são desacoplados dos componentes de interface de usuário. A comunicação entre essas camadas ocorre por meio de APIs baseadas em protocolos interoperáveis, normalmente utilizando serviços REST sobre HTTP/HTTPS.

Essa abordagem favorece a independência tecnológica entre frontend e backend, permitindo maior flexibilidade na evolução das aplicações e melhor escalabilidade da solução.

2.3. Público-Alvo

Este documento tem como público-alvo órgãos e unidades parceiras, arquitetos de software, desenvolvedores de sistemas, gerentes de projeto e demais profissionais envolvidos no ciclo de vida de desenvolvimento e manutenção dos sistemas institucionais.

2.4. Benefícios Arquiteturais

A adoção da arquitetura de referência definida neste documento proporciona diversos benefícios técnicos e organizacionais, contribuindo para o desenvolvimento de sistemas mais seguros, escaláveis, padronizados e com melhor desempenho.

✓ **Segurança**

A arquitetura proposta incorpora mecanismos de segurança baseados em padrões amplamente adotados no mercado, garantindo proteção adequada aos sistemas e aos dados institucionais.

✓ **Escalabilidade**

A arquitetura é projetada para facilitar a evolução e expansão dos sistemas ao longo do tempo, permitindo que novas funcionalidades sejam adicionadas com baixo impacto nos componentes existentes.

✓ **Padronização**

A arquitetura de referência estabelece padrões técnicos e organizacionais que promovem maior consistência entre os projetos desenvolvidos pela instituição.

✓ **Performance**

A arquitetura adota tecnologias modernas de build e otimização, contribuindo para aplicações mais rápidas e eficientes.

3. Modelo de Arquitetura

Model View Controller (MVC): esse design pattern separa a lógica da aplicação da interface do usuário final. O padrão consiste na divisão em 3 (três) camadas: Model, View e Controller.

3.1. Framework MVC

O padrão Model-View-Controller (MVC) é uma forma de organizar o desenvolvimento de software por meio da separação de responsabilidades em três componentes principais: Model, View e Controller.

Quando aplicado em arquiteturas modernas baseadas em APIs e aplicações web desacopladas, esse conceito permite estruturar o sistema de forma mais organizada e escalável. Nesse modelo, o backend, implementado com Spring Boot, concentra as responsabilidades relacionadas ao Model e ao Controller, incluindo a lógica de negócio, regras de validação e exposição de APIs REST.

A camada View é implementada no frontend, utilizando tecnologias modernas como React, responsáveis pela construção da interface de usuário e pela interação com os serviços disponibilizados pelo backend.

Essa separação entre backend (server-side) e frontend (client-side) promove maior desacoplamento entre as camadas da aplicação, facilitando a manutenção, evolução tecnológica e escalabilidade dos sistemas.

3.1.1. Camada de Modelo (Model)

Função

Representa as regras de negócio, a lógica central da aplicação e os mecanismos de persistência de dados, sendo responsável pelo processamento das informações e pela garantia da integridade das operações do sistema.

No Backend

Essa camada é implementada no backend da aplicação, utilizando a plataforma Java com o framework Spring Boot, responsável por organizar os componentes de negócio, serviços e acesso a dados.

Responsabilidades

- Implementação das regras de negócio da aplicação.
- Validação e manipulação de dados provenientes das requisições dos usuários ou de integrações externas.
- Acesso e persistência de dados, por meio de mecanismos de integração com bancos de dados corporativos, como Oracle Database.
- Exposição de APIs REST, utilizando o formato JSON, que disponibilizam dados e funcionalidades para consumo pela camada de interface (frontend).

3.1.2. Camada de Visualização (View)

Função

É a camada responsável pela interface com o usuário final, permitindo a visualização das informações do sistema e a interação com suas funcionalidades.

No Frontend

Na arquitetura proposta, a camada de visualização é implementada utilizando React, com apoio de TypeScript, responsável pela construção de interfaces modernas baseadas em componentes reutilizáveis. A interface segue os princípios do Material Design, garantindo consistência visual e melhor experiência de uso.

Responsabilidades

- Renderizar informações provenientes das APIs disponibilizadas pelo backend.
- Apresentar a interface do usuário, incluindo componentes visuais, formulários, navegação e elementos interativos.
- Capturar ações do usuário, como cliques, preenchimento de formulários e eventos de navegação.
- Realizar requisições ao backend, por meio de chamadas a APIs REST, para envio ou consulta de dados.

3.1.3. Controller (Camada de Mediação / Orquestração)

Função

A camada de Controller atua como intermediária entre a camada de visualização (View) e a camada de modelo (Model), sendo responsável por receber as requisições da aplicação cliente, encaminhá-las para o processamento da lógica de negócio e retornar as respostas apropriadas.

No Backend

Na arquitetura proposta, os Controllers são implementados no backend utilizando Spring Boot, sendo responsáveis por receber requisições HTTP provenientes do frontend React, processá-las e encaminhá-las para os componentes responsáveis pela lógica de negócio. Esses componentes expõem endpoints de APIs REST, que permitem a comunicação entre a interface do usuário e os serviços da aplicação.

Responsabilidades

- Receber requisições HTTP provenientes da interface do usuário ou de outros sistemas integrados.
- Encaminhar as requisições para os serviços responsáveis pela execução das regras de negócio.
- Orquestrar o fluxo de processamento, garantindo a correta interação entre as camadas da aplicação.
- Retornar respostas estruturadas, normalmente no formato JSON, para consumo pela camada de apresentação (frontend).

3.2. Padrões de Arquitetura

O sistema é desenvolvido seguindo o modelo arquitetural cliente-servidor, estruturado em camadas bem definidas que promovem separação de responsabilidades, desacoplamento entre componentes e maior facilidade de manutenção e evolução da aplicação.

A arquitetura da solução é composta por três camadas principais: camada de apresentação (frontend), camada de serviços (backend) e camada de integração e persistência de dados.

A camada de apresentação é implementada utilizando React com TypeScript, com suporte da ferramenta de build Vite, sendo responsável pela construção da interface do usuário e pela interação com os serviços disponibilizados pelo backend.

A camada de serviços é implementada na plataforma Java, utilizando o framework Spring Boot, responsável por expor APIs REST que concentram a lógica de negócio da aplicação, além de realizar o processamento das requisições provenientes do frontend e de sistemas externos.

A camada de persistência e integração é responsável pelo acesso aos dados e pela comunicação com outros sistemas corporativos, incluindo a integração com bancos de dados institucionais, como o Oracle Database, e serviços externos quando necessário.

A comunicação entre o frontend e o backend ocorre por meio de serviços REST sobre HTTP/HTTPS, utilizando JSON como formato padrão de troca de dados, garantindo interoperabilidade entre os componentes da solução.

4. Frontend

O frontend da aplicação é desenvolvido como uma Single Page Application (SPA) utilizando React, com suporte da linguagem TypeScript e da ferramenta de build Vite. Essa abordagem permite a construção de interfaces modernas, dinâmicas e altamente interativas, proporcionando melhor experiência ao usuário e maior eficiência na comunicação com os serviços do backend.

O React adota uma arquitetura baseada em componentes, na qual cada componente encapsula sua própria estrutura de interface, comportamento e estilos, promovendo modularização, reutilização de código e maior organização do projeto.

Os componentes são responsáveis por representar partes específicas da interface do usuário e podem ser combinados para formar interfaces mais complexas, facilitando a manutenção e evolução da aplicação ao longo do tempo.

A interface segue os princípios do Material Design, garantindo padronização visual, consistência de interface e melhor usabilidade nos sistemas desenvolvidos.

A comunicação com o backend ocorre por meio de requisições a APIs REST, utilizando HTTP/HTTPS e JSON como formato padrão de troca de dados.

4.1. Visão Geral

A arquitetura do Frontend é baseada nos princípios modernos adotados internacionalmente:

- Feature-Sliced Architecture (FSA)
- Clean Architecture adaptada para frontend
- Domain-driven UI
- Separation of Concerns
- Scalability-first Design

Essa combinação garante:

✓ Alta escalabilidade

✓ Baixo acoplamento

✓ Extensibilidade

✓ Reuso consistente

✓ Testabilidade

✓ Aderência a padrões globais

4.2. Principais Tecnologias

Tecnologia	**Versão**	**Função**
React 19	19.x	Framework SPA, React Compiler-ready
TypeScript 5.7	5.7	Tipagem e segurança
Vite 6	6	Build e Dev Server modernos
Keycloak	26	Autenticação e Autorização (OIDC)
Axios	-	HTTP Client com interceptors
React Router 7	7	Navegação
React Hook Form	-	Gerenciamento de formulários
Vitest + Testing Library	-	Testes unitários
GOVBR DS Core	-	Design System oficial do Governo Federal

4.3. Estrutura de Pastas

```
src
├── app
│   ├── App.tsx
│   ├── main.tsx
│   ├── router/       # definição das rotas da aplicação
│   ├── config/       # configurações globais
│   └── providers/    # providers globais (contextos, tema, etc.)
│
├── shared
│   ├── components/   # Design System ou wrappers do GOVBR DS
│   ├── hooks/        # hooks reutilizáveis
│   ├── utils/        # funções utilitárias
│   ├── types/        # tipos globais
│   └── style/        # estilos globais
│
├── features          # organização por domínio de negócio
│   ├── auditor
│   │   ├── pages/
│   │   ├── components/
│   │   ├── services/
│   │   ├── hooks/
│   │   ├── state/
│   │   └── index.ts
│   ├── gestor
│   ├── administrador
│   └── ...           # cada domínio vira uma feature
│
├── services
│   ├── http
│   │   ├── HttpProvider.ts
│   │   ├── interceptors.ts
│   │   └── api-client.ts
│   ├── auth
│   │   └── msal.ts
│   └── external
│       └── govbr/
│
├── assets
│   ├── img/
│   └── icons/
│
└── tests
```

4.4. Camadas da Arquitetura

Camada App (Core Layer)

Responsável por:

- Bootstrap da aplicação
- Configurações globais
- Providers (keycloak, router, theme)
- Env & config

- Error boundaries

Elementos principais: main.tsx, App.tsx, router/routes.tsx, providers/keycloak.

Shared (Cross-cutting Layer)

Responsável por tudo que é genérico e reutilizável, como:

- Componentes atômicos / GOVBR wrappers
- Hooks genéricos
- Tipos globais
- Funções utilitárias
- Estilos

Essa camada não depende de features.

Features (Domain Layer)

Cada domínio funcional fica isolado (ex.: features/auditor/, features/gestor/, features/administrador/). Cada feature contém:

- Páginas (UI)
- Componentes específicos
- Services
- Hooks
- Tipos

Infra Layer (services/http)

Contém:

- Axios configurado
- Interceptores
- Retry logic
- Serializers/adapters
- Keycloak token injection

Autenticação e Autorização (Keycloak)

A arquitetura usa Keycloak para:

- Login
- Logout
- Refresh automático do token
- RBAC via roles do realm e client
- Atributos customizados do usuário

Componentes principais: keycloak.ts (instancia o client), KeycloakProvider.tsx (gerencia sessão), keycloak-auth.ts (abstrações auxiliares).

Serviços por Feature

Exemplo: features/auditor/services/auditor.service.ts. Cada serviço:

✓ Implementa apenas o domínio

✓ Não manipula UI

✓ Não acessa diretamente outros domínios

Padrões de Componentização

A arquitetura segue:

- Componentes atômicos (shared/components)
- Componentes de domínio (features/*/components)
- Layout como componente raiz
- Reutilização orientada a design system

Padrões de Código

TypeScript estrito; ESLint + TS + React Hooks Rules; formatação via Prettier (recomendado); import aliases (ex.: @/shared/utils). Padrão de nome de arquivo:

- Componente React → PascalCase
- Funções/utilidades → camelCase
- Tipos → PascalCase + .types.ts

Testes

Stack: Vitest, Testing Library, jsdom, cobertura via vitest --coverage.

O que testar:

- Componentes puros
- Hooks
- Serviços
- Lógica de domínio

5. Backend

A aplicação backend deverá adotar uma arquitetura em camadas (Layered Architecture), organizada de forma a separar claramente as responsabilidades entre os diferentes níveis da aplicação, favorecendo manutenibilidade, testabilidade, escalabilidade e resiliência.

A solução será implementada no estilo arquitetural de microserviços, utilizando Java 21 e o framework Spring Boot 4.0.3, responsável por fornecer a infraestrutura base para desenvolvimento de aplicações corporativas baseadas no ecossistema Spring.

O Spring Boot simplifica a configuração e inicialização das aplicações por meio do conceito de starters, que são conjuntos pré-configurados de dependências interoperáveis. Esses starters encapsulam

bibliotecas e configurações necessárias para determinadas funcionalidades (como persistência, segurança, mensageria e APIs REST), permitindo que a aplicação seja construída seguindo o princípio de convenção sobre configuração (Convention over Configuration).

A arquitetura do backend deverá ser estruturada nas seguintes camadas pilares:

Application Layer

Responsável por orquestrar os casos de uso da aplicação, coordenando serviços de domínio, validações de fluxo e regras de aplicação.

Domain Layer

Contém o núcleo do domínio de negócio, incluindo entidades, agregados, objetos de valor (Value Objects) e regras de negócio fundamentais, seguindo princípios de Domain-Driven Design (DDD).

Infrastructure Layer

Responsável pela integração com tecnologias externas, como repositórios de persistência, sistemas de mensageria (ex.: Kafka), APIs externas e mecanismos de acesso a dados.

Os microserviços expõem suas funcionalidades por meio de APIs REST, que são consumidas pelo frontend e por outros serviços, mantendo a comunicação baseada em contratos JSON.

Essa organização promove baixo acoplamento entre camadas, permitindo evolução independente dos componentes e facilitando a escalabilidade da plataforma.

5.1. Visão Geral

O backend é uma aplicação Java moderna construída sobre:

- Java versão 21
- Spring Boot 4.0.3
- Maven
- Configuração Spring Cloud (bootstrap.yml)

Deploy via:

- Docker
- Kubernetes (Helm)
- Azure DevOps Pipelines (YAML)

A arquitetura foca em:

- Modularidade
- Separação de responsabilidades
- Uso de camadas bem definidas
- Escalabilidade
- Compatibilidade com microsserviços

5.1. Tecnologias do Backend

5.1.1. Principais Tecnologias

Categoria	Tecnologias	Finalidade
Linguagem	Java 21 LTS	Suporte a Virtual Threads (Loom), Records, Pattern Matching, performance moderna
Framework Principal	Spring Boot 4.0.3	Framework líder mundial para APIs Java, cloud-native
Arquitetura	Clean Architecture, MVC, DDD	Camadas claras, independência de frameworks, escalabilidade
Build	Maven 3.x (mvnw wrapper)	Build padronizado, reprodutibilidade
Configuração	application.yml / bootstrap.yml	Configuração centralizada, profiles, Cloud Config (opcional)

5.1.2. Segurança e Identidade

Área	Tecnologias	Finalidade
Autenticação	Keycloak 26 (OIDC/OAuth2)	Identity Provider central
Autorização	Spring Security 6	Controle de acesso, roles, policies
Tokens	JWT (Access/Refresh)	Autenticação stateless
Secret Management	Kubernetes Secrets / Azure Key Vault	Segredos seguros

5.1.3. Persistência e Dados

Categoria	Tecnologias	Finalidade
ORM	Spring Data JPA + Hibernate	Persistência relacional
Banco	Oracle Database 19c	Banco relacional corporativo — padrão primário
Banco (alternativo)	PostgreSQL, SQL Server	Bancos relacionais admitidos conforme ambiente de hospedagem ou legado do projeto
Busca	Elasticsearch (opcional)	Busca full-text
Big Data / Analytics	Databricks	Processamento de dados em larga escala e cargas analíticas

5.1.4. Infraestrutura / Cloud Native

Área	Tecnologias	Finalidade
Containers	Docker	Empacotamento da aplicação
Orquestração	Kubernetes/AKS + Helm	Deploy padronizado e escalável
CI/CD	Azure Pipelines	Build, testes, análise, deploy
Armazenamento de arquivos	Azure Blob Storage	Upload/stream de arquivos
API Gateway	API Management Service	Interceptação segura dos endpoints internos

5.1.5. Qualidade, Testes e Automação

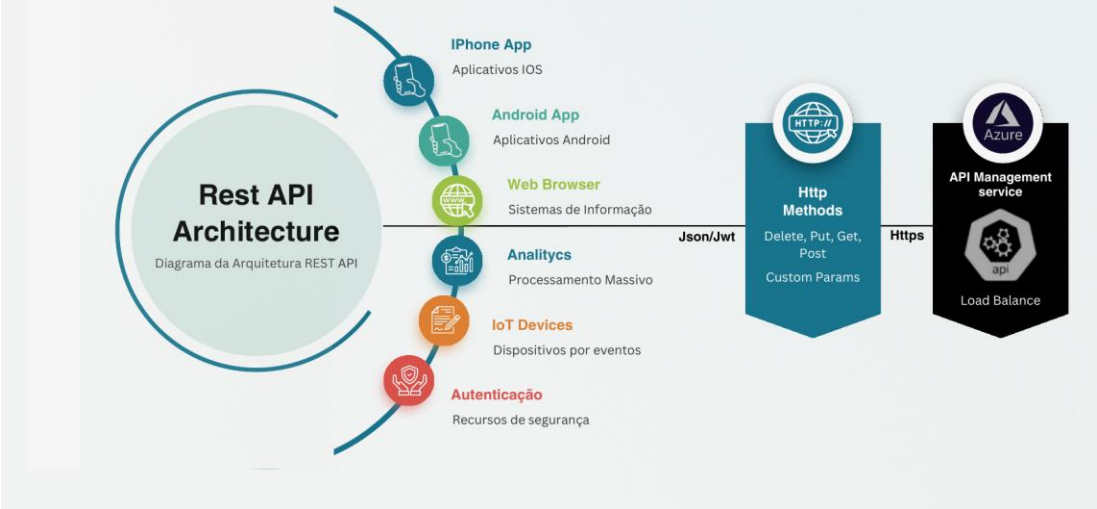
Categoria	Tecnologias	Finalidade

Testes unitários	JUnit 5	Testes unitários modernos
Lint/Análise estática	SonarQube	Governança e qualidade
Test Automation	Robot Framework 7.4.2	Testes automatizados na CI/CD

5.2. Estrutura de Pastas

```
project-root
│
├── devops/                    # Artefatos de infraestrutura e entrega contínua
│   ├── ci-cd/                 # Pipelines de integração e entrega contínua
│   │   └── azure-pipeline.yml # (exemplo de pipeline)
│   ├── container/             # Configuração de containers
│   │   ├── Dockerfile
│   │   └── rootfs/            # Estrutura base do container (quando aplicável)
│   ├── orchestration/         # Configurações de orquestração (ex.: Kubernetes)
│   │   └── values.yaml        # (exemplo de configuração de deployment)
│   └── README.md
│
├── build/                     # Artefatos gerados durante o processo de build
├── scripts/                   # Scripts utilitários de build ou automação (opcional)
│
├── pom.xml                    # Configuração de dependências e build (Maven)
├── mvnw                       # Maven Wrapper
├── mvnw.cmd
├── sonar-project.properties   # Configuração de análise estática de código (exemplo)
│
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── org/
│   │   │       └── exemplo/          # Namespace base da aplicação
│   │   │           └── application/
│   │   │               ├── api/            # Camada de exposição (REST Controllers)
│   │   │               ├── application/    # Camada de aplicação (casos de uso)
│   │   │               ├── domain/         # Camada de domínio (entidades e regras)
│   │   │               ├── infrastructure/ # Camada de infraestrutura
│   │   │               │                   # (persistência, integrações externas)
│   │   │               └── config/         # Configurações da aplicação
│   │   └── resources/
│   │       ├── application.yml   # Configurações da aplicação
│   │       ├── bootstrap.yml     # Configurações de inicialização (exemplo)
│   │       └── static/           # Recursos estáticos (quando aplicável)
│   └── test/
│       └── java/
│           └── org/
│               └── exemplo/      # Estrutura de testes automatizados
│
└── README.md
```

5.3. Diagrama Arquitetura REST API



5.4. Camadas da Arquitetura

Controller (API Layer)

- Implementa endpoints REST
- Faz validação básica
- Não contém regra de negócio
- Expõe DTOs (entrada/saída)
- Integra com Swagger/OpenAPI

Service (Application Layer)

- Contém regras de negócio
- Orquestra chamadas ao domínio
- Chama repositórios
- Aplicação de casos de uso

Domain (Domain Layer)

Coração do negócio. Contém:

- Entidades
- Agregados
- Value Objects
- Domain Services
- Domain Events

Não depende de frameworks. Regras de negócio puras.

Repository / Adapter (Infrastructure Layer)

- Implementa as portas de saída (ports) definidas no domínio
- Acesso ao banco de dados (Spring Data JPA, JDBC, QueryDSL)

- Conexão com sistemas externos (HTTP clients, mensageria)
- Gestão de persistência, cache, filas, arquivos
- Implementações concretas: apoio tecnológico, não lógica de negócio

Config (Cross-Cutting Layer)

- Configurações do Spring Boot
- Beans reutilizáveis
- Segurança (Keycloak, JWT)
- Serialização/Deserialização (Jackson)
- Configuração de logs, CORS, profiles
- Filters e interceptors técnicos

Exception Handling

- Camada centralizada para padronizar erros da API
- Uso recomendado: RFC 9457 (Problem Details)
- Mapeamento de erros técnicos e erros de domínio

6. Considerações Finais

6.1. Boas Práticas

- Criar feature modules independentes, carregados de forma lazy loading para otimizar performance.
- Colocar services globais no CoreModule e services específicos dentro do módulo de funcionalidade correspondente.
- Reaproveitar componentes pequenos (botões, inputs, modais) no SharedModule.
- Usar environment.ts para separar configurações de dev, homolog e produção.

6.2. Convenções de Código

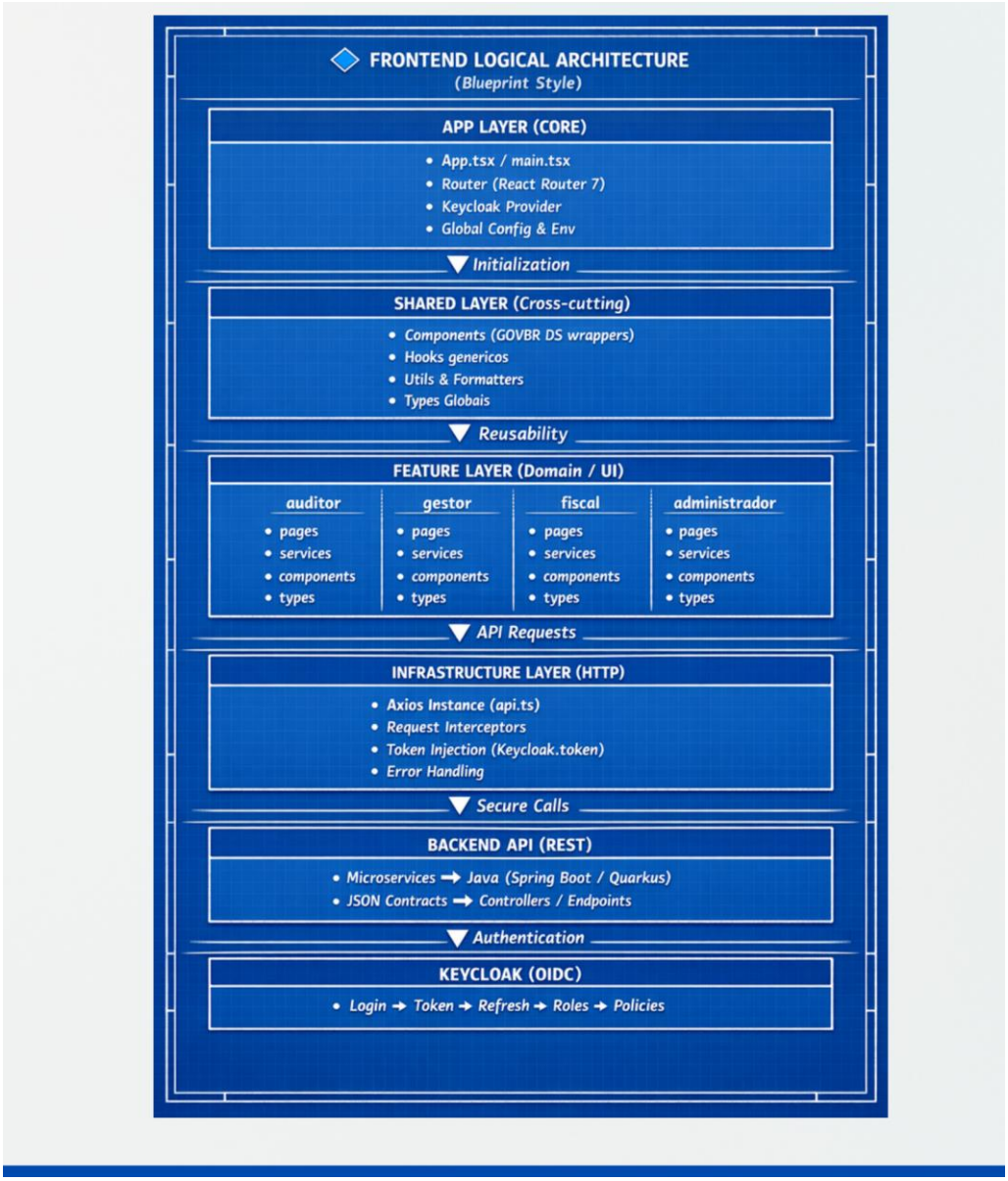
É extremamente recomendável o uso de boas práticas e das convenções das tecnologias adotadas no projeto.

A qualidade do código deve ser constantemente verificada com auxílio do SonarQube, utilizando as métricas aplicadas com profile disponibilizado pelo Ministério. O projeto Backend possuirá mecanismo de integração com o pacote de software Nextcloud. Imagens e arquivos importantes utilizados em regras negociais implementadas nos projetos serão hospedados nesta plataforma.

Convenção Java: https://google.github.io/styleguide/javaguide.html

Convenção Projetos Spring Java: https://github.com/spring-projects/spring-framework/wiki/Code-Style

7. Diagrama Lógico da Arquitetura de Referência



Documento técnico original elaborado pela Coordenação-Geral de Soluções de Tecnologia da Informação (CGTEC) / Coordenação de Arquitetura, Qualidade, Testes e Automação (CAQ) — STI/SE/MJSP.