Arquitetura Corporativa

Sistemas WEB

1. INTRODUÇÃO

O Departamento de Tecnologia da Informação (DTI) do MCTIC é responsável por disponibilizar e manter serviços de tecnologia essenciais às áreas de negócio e aos usuários finais. Esses serviços são suportados por processos, aplicações e infraestrutura que precisam operar com altos índices de disponibilidade, desempenho e qualidade.

Nesse contexto, a área de Tecnologia da Informação (TI) do MCTIC evoluiu a sua arquitetura corporativa de sistemas de informação estruturando soluções que visam a reutilização de serviços através de uma Arquitetura Orientada a Serviços (SOA), conforme descrito nas seções seguintes deste documento.

Todo o arcabouço tecnológico de aplicações do MCTIC é baseado em um ambiente distribuído e escalável de microserviços, que estão disponibilizados em um Enterprise Service Bus (ESB) e orquestrados por ferramentas Business Process Management Suite (BPMS). Através de um ambiente DevOps, existe um pipeline que habilita práticas como a integração e distribuição contínua, microserviços, infraestrutura como código (contêiner), gerenciamento da configuração, políticas como código, monitoramento e registro em log, entre outras práticas de construção, testes e implantação de forma contínua, ou seja, atualmente o MCTIC possui um ambiente de sistemas de informação dinâmico e em constante evolução.

Além disso, como os projetos do MCTIC possuem diversas integrações com serviços/sistemas (webservices) internos e externos, há uma necessidade de manter a interoperabilidade entre esses serviços/aplicações de forma eficiente devido ao crescente volume de requisições de acesso e transações de dados.

O fato do MCTIC ser responsável e principal fomentador de tecnologia no mercado nacional, estando diretamente ligado às políticas e ações que visam o avanço da ciência, tecnologia e inovação no País, aumenta a responsabilidade do MCTIC no desenvolvimento de sistemas de informação que são utilizados pelos cidadãos alinhados com as principais tecnologias inovadoras que são padrões de mercado.

Por fim, ressalte-se também que, além dos processos existentes, toda essa Arquitetura de Referência foi desenvolvida pelo próprio MCTIC com base nas melhores práticas de mercado. Tendo como resultado um arcabouço tecnológico de propriedade do MCTIC, pois foi desenvolvido, implantado e melhorado pelo próprio órgão. Além disso, tanto os Processos quanto à Arquitetura de Referência estão em constante evolução pela Coordenação-geral de Sistemas (CGSI) para garantir a execução dos projetos de acordo com os padrões de mercado, com qualidade satisfatória e sempre em busca de inovação.

2. SOLUÇÃO ARQUITETURAL

Esta seção tem o objetivo de apresentar a solução arquitetural corporativa para sistemas de informação definida pelo MCTIC, ou seja, descrever as características que orientam a arquitetura, bem como dar uma visão geral sobre a estrutura das camadas, pacotes, estereótipos de classe, bem como apresentar as demais restrições e motivações que subsidiam a sua construção.

Dessa forma, ao compreender os elementos que compõem a arquitetura, as funções e a arquitetura de cada tecnologia, possibilitará o melhor entendimento dos guias operacionais que orientam a construção de código.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Visão geral da solução arquitetural.
- Restrições arquiteturais significantes.
- Requisitos não funcionais de qualidade:
    - Interoperabilidade;
    - Usabilidade;
    - Manutenibilidade;
    - Confiabilidade, Escalabilidade e Disponibilidade Performance.

2.1. Visão geral da solução arquitetural

Visando apresentar todos os elementos e tecnologias utilizadas com base nas restrições e requisitos do MCTIC, conforme as "Restrições arquiteturais", a ***Figura 1*** ilustra uma visão geral da solução arquitetural que foi especificada.

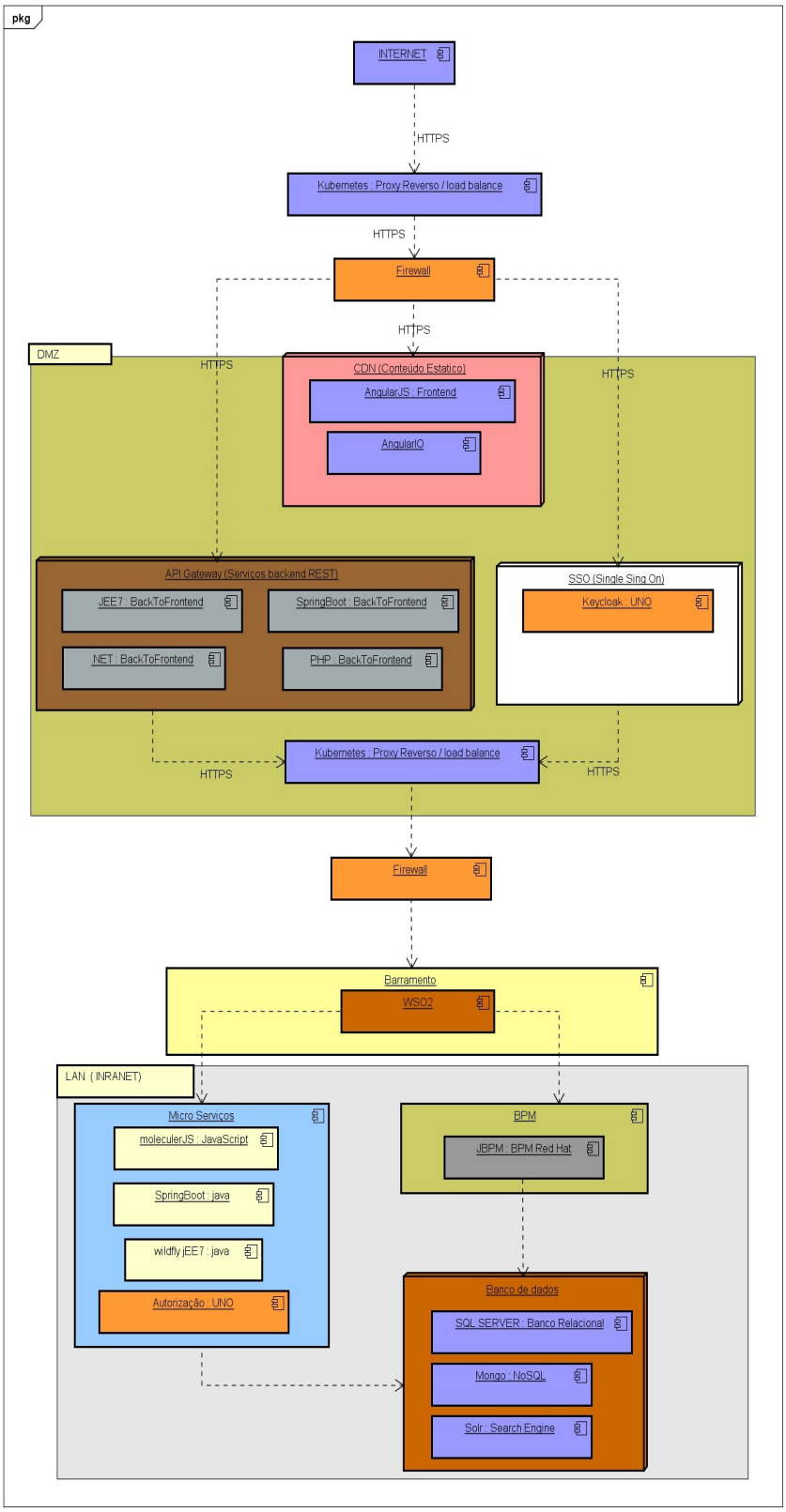


Figura 1 - Solução arquitetural

2.2. Restrições arquiteturais

Visando atender as necessidades do MCTIC, algumas restrições tiveram que ser observadas na instanciação da solução arquitetural. A Tabela 1 - Restrições do cliente descreve uma relação exemplificativa de restrições que foram levadas em consideração para a solução arquitetônica ilustrada na Figura 2.

Restrições significativas para a solução da arquitetura	
1	Possibilidade de utilização de múltiplas linguagens de programação, tais como: Java, JavaScript, .NET, PHP, entre outras.
2	Os sistemas devem ser responsivos para serem acessados por vários tipos de dispositivos via WEB.
3	Os sistemas podem ter um aumento significativo de acesso de usuários.
4	Os sistemas devem responder as requisições em menos de 3 segundos em 99% dos casos.
5	Os Processos de negócio dos sistemas devem possuir facilidade para evoluções continuas.
6	O Login deverá centralizado em apenas um local, e poder adicionar múltiplos provedores de identidade tais como: banco LDAP, Banco de dados etc.
7	Os dados corporativos devem ser coesos e de fácil evolução e correção.
8	Os sistemas devem possuir a flexibilidade de se comunicar com banco de dados estruturados, semi estruturados e não estruturados.

Tabela 1 - Restrições do cliente

Essas restrições, também chamadas de ***requisitos não funcionais de qualidade***, podem ser divididas nas categorias descritas na Tabela 2 - Requisitos não funcionais.

Categoria	**Descrição**

Interoperabilidade	Habilidade dos sistemas trocarem informações entre si com tecnologias heterogêneas.
Usabilidade	Facilidade com que as pessoas podem empregar uma ferramenta ou objeto a fim de realizar uma tarefa específica e importante.
Manutenibilidade	É a facilidade em manter o sistema, como por exemplo para correção de bugs e evoluções.
Confiabilidade	Habilidade do sistema sempre se comportar da maneira esperada.
Desempenho	Habilidade do sistema atender os objetivos dentro do tempo proposto.
Portabilidade	Capacidade de ser compilado ou executado em diversos hardwares e sistemas operacionais.
Reusabilidade	Funcionalidades que podem ser utilizadas de uma maneira fácil por outros sistemas.
Segurança	Habilidade do sistema proteger os dados.

Tabela 2 - Requisitos não funcionais

Requisito não funcional	**Restrições**	**Solução**
Interoperabilidade	• Possibilidade de uso de múltiplas linguagens de programação, tais como: Java, JavaScript, .NET, PHP, entre outras; • Os sistemas devem possuir a flexibilidade de se comunicar com banco de dados relacionais e não relacionais (NoSql); - Os sistemas devem	Interoperabilidade

	ser responsivos para serem acessados por vários tipos de dispositivos via WEB.	
Usabilidade	Os sistemas devem ser responsivos para serem acessados por vários tipos de dispositivos via WEB.	Usabilidade
Manutenibilidade	• Os dados corporativos devem ser coesos e de fácil mudança; • Os Processos de negócio dos sistemas devem possuir facilidade para evoluções contínuas.	Manutenibilidade
Confiabilidade, Escalabilidade ou Disponibilidade	Os sistemas podem ter um aumento significativo de acesso de usuários.	Confiabilidade, Escalabilidade ou Disponibilidade
Segurança	O Login deverá ser centralizado em apenas um local, e poder adicionar múltiplos provedores de identidade tais como banco LDAP, Banco de dados etc.	Segurança
Performance	Os sistemas devem responder as requisições em menos de 3 segundos em 99% dos casos.	Performance

Tabela 3 - Relação entre as restrições, requisitos não funcionais e solução arquitetural

2.3. Requisitos não funcionais de qualidade

2.3.1. Interoperabilidade

Dentre os requisitos não funcionais que tangem interoperabilidade, pode-se destacar:

- Utilização de múltiplas linguagens de programação, tais como: Java, JavaScript, .NET, PHP, dentre outras;
- Os sistemas devem possuir a flexibilidade de se comunicar com banco de dados relacionais e não relacionais (NoSql);
- Os sistemas devem ser responsivos para serem acessados por vários tipos de dispositivos via WEB.

Para atender os requisitos citados acima optou-se em utilizar o estilo de arquitetura REST. Esse conjunto de princípios utiliza o protocolo HTTP como meio de transporte para troca informações através do tipo de mídia. O HTTP é o protocolo de comunicação utilizado pelos browsers e também por todas as linguagens de programação utilizadas no MCTIC, garantindo assim a interoperabilidade entre os sistemas.

2.3.2. Usabilidade

Dentre os requisitos não funcionais que tangem a usabilidade, pode-se destacar:

- Os sistemas devem ser responsivos para serem acessados por vários tipos de dispositivos via WEB

Para atender esse requisito inicialmente foi escolhida pelo MCTIC a especificação de interfaces MATERIAL.IO, que foi criada pelo Google e possui diversas regras que estão extremamente alinhadas a usabilidade e responsividade para diversos tipos de dispositivos. Para essa especificação o framework definido para o desenvolvimento de Interfaces web foi o AngularJS, por utilizar a arquitetura SPA (Single Page Application). Por fim, o framework de componentes que foi definido foi o MaterialJS por implementar a especificação do MATERIAL.IO.

Como recentemente o framework AngularJS teve uma grande evolução, originando um novo framework chamado de Angular.IO, que é incompatível com a versão 1.x. (AngularJS). Dessa forma, os novos sistemas do MCTIC estão sendo criados utilizando o framework AngularIO.

2.3.3. Confiabilidade / Escalabilidade / Disponibilidade

Dentre os requisitos não funcionais que tangem a Confiabilidade, Escalabilidade e Disponibilidade, pode-se destacar:

- Os sistemas podem ter um aumento significativo de acesso de usuários;
- Os sistemas devem responder as requisições em menos de 3 segundos em 99% dos casos.

Para garantir esse requisito a arquitetura utiliza uma estratégia de balanceamento de carga (Load Balance - LB) com o algoritmo Round-Robin para enviar as requisições ao backend.

Como está sendo utilizada a arquitetura REST, nenhuma aplicação mantém estado, ou seja, o estado é mantido em um token JWT. Assim toda chamada a qualquer nó do cluster deve obter o mesmo resultado.

Dessa forma, como a aplicação não mantém estado, é possível escalonar verticalmente ou horizontalmente qualquer CDN, API gateway ou microserviço.

A arquitetura conta também com os microserviços para dados e serviços que são compartilhados (reutilizados) por diversas aplicações, é possível realizar o escalonamento com um menor nível de granularidade.

2.3.4. Segurança

Dentre os requisitos não funcionais que tangem a segurança, pode-se destacar:

- O Login deverá centralizado em apenas um local;
- Possibilitar o uso de múltiplos provedores de identidade tais como banco LDAP, Banco de dados, entre outros.

Para atender esse requisito foi escolhido o produto open source Keycloack, que possibilita a criação de plugins para conectar em múltiplas fontes de dados. O Keycloack também possui a implementação de diversos protocolos de autenticação compatíveis com arquitetura REST, tais como: Auth0, Auth2, Open ID, Open ID Connect, entre outros.

2.3.5. Performance

Dentre os requisitos não funcionais que tangem a performance, pode-se destacar:

- Os sistemas devem responder as requisições em menos de 3 segundos em 99% dos casos;
- A aplicação pode ser escalonada horizontalmente e o Load Balance deverá dividir a carga de requisições para evitar a lentidão em horários de pico.

Para consultar grandes volumes de dados pode ser utilizado um indexador de dados Solr ou ElasticSearch, ou a otimização de consultas de banco de dados.

Para melhorar o tempo de requisição e resposta, é utilizado o padrão "Backend to Frontend". No exemplo abaixo será descrita a realização do padrão, para um serviço de recuperar dados do usuário cadastrado no sistema XPTO.

Dados que a interface irá recuperar:

- Número protocolo do sistema XPTO (Sistema Monolítico);
- Matrícula do usuário (Microserviço);
- CPF do usuário (Microserviço);
- Endereço do usuário logado (Microserviço);
- UF (Microserviço).

Para exemplificar, A Figura 2 ilustra a implementação SEM UTILIZAR o padrão de projeto "Backend To Frontend".

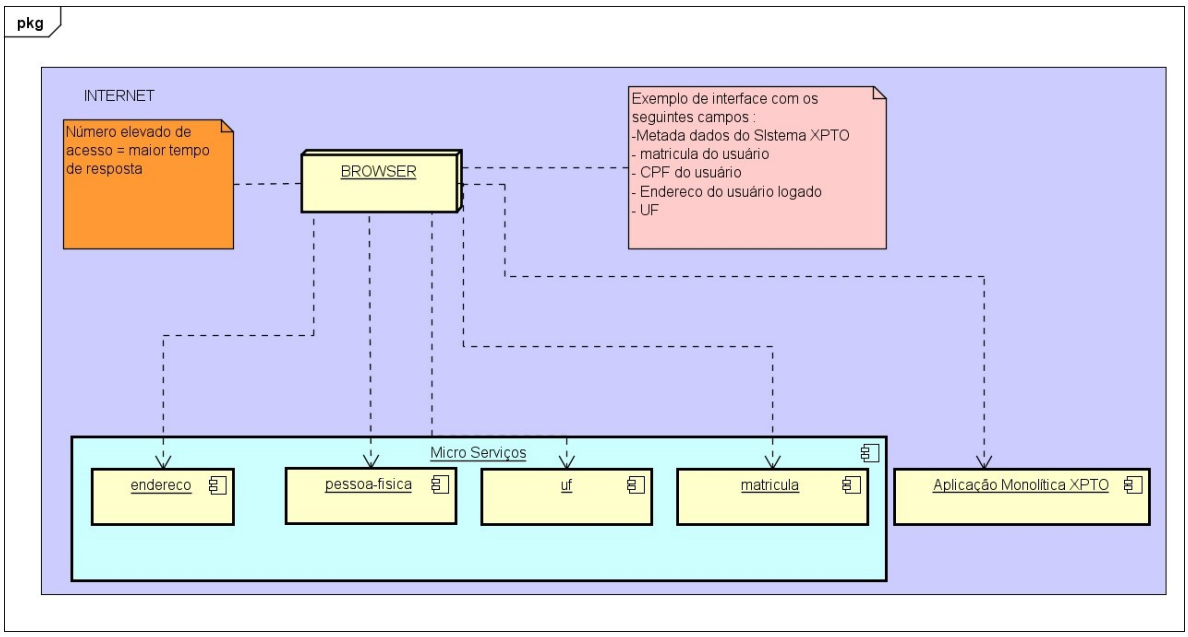


Figura 2 - implementação SEM UTILIZAR o padrão de projeto "Backend To Frontend"

Problema: Cinco requisições utilizando extranet com uma latência maior resultando da diminuição da performance.

Por outro lado, A Figura 3 ilustra a Implementação UTILIZANDO o Padrão de projeto "Backend To Frontend".

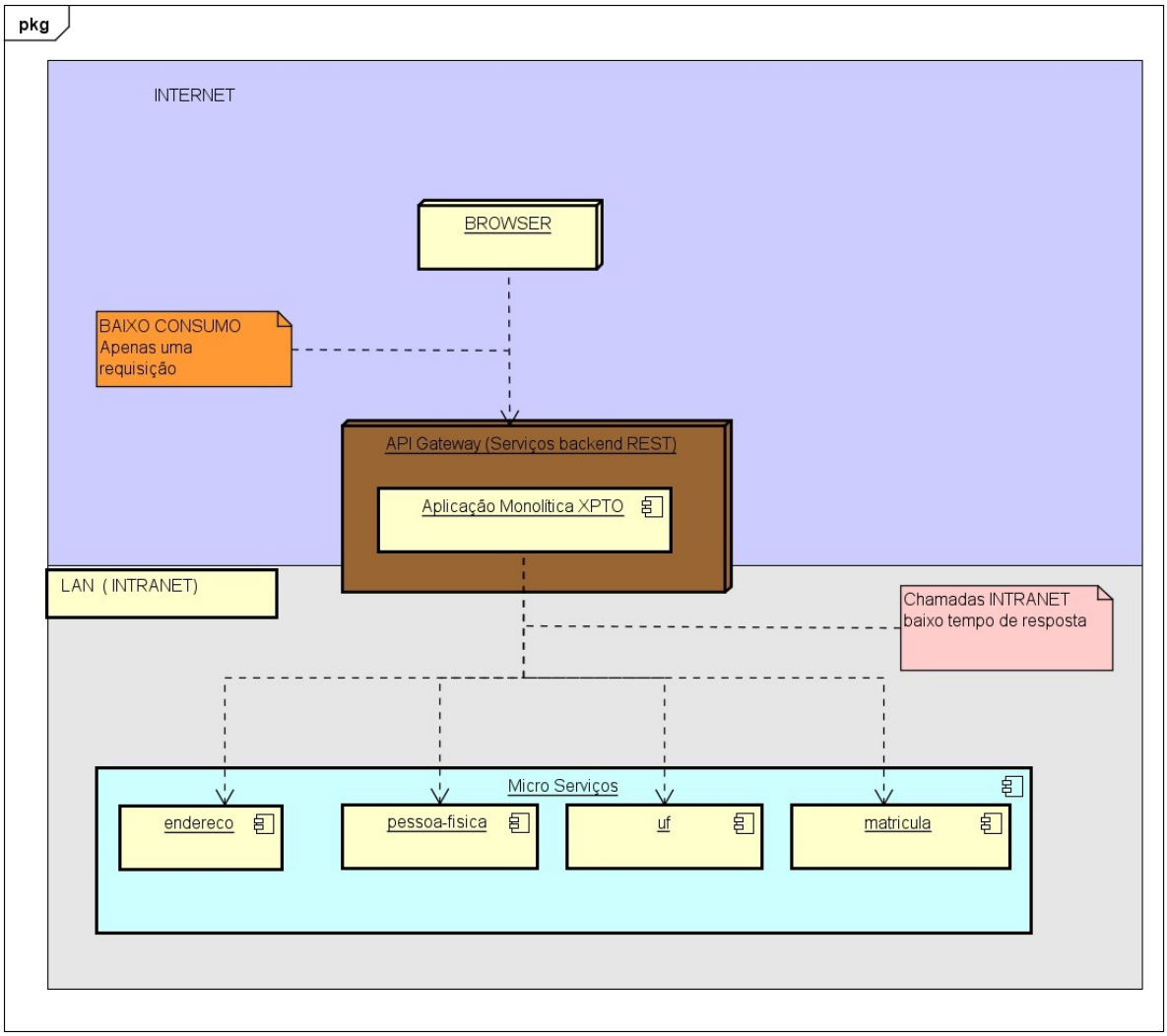


Figura 3 - Implementação UTILIZANDO o Padrão de projeto "Backend To Frontend"

Solução: Utilizar o padrão BackEnd To Frontend para o usuário externo realizar apenas uma requisição e as outras chamadas serão realizadas via intranet que tem uma latência menor.

2.3.6. Manutenibilidade

Dentre os requisitos não funcionais que tangem a Manutenibilidade, pode-se destacar:

- Os dados corporativos devem ser coesos, de fácil evolução e correção;
- Os Processos de negócio dos sistemas devem possuir facilidade para evoluções contínuas.

Para manter os dados corporativos coesos e de fácil manutenção foi adicionada a estratégia de utilização de microserviços na solução arquitetural, ou seja, os microserviços por serem pequenos são mais fáceis de alteração da tecnologia, realizar testes, realizar entregas e evoluções contínuas.

Para tratar a automação dos processos de negócio foi escolhida a solução de ***Business Process Modelling Suite (BPMS)*** JBPM, que permite que seja feita a modelagem de metas de negócios descrevendo as etapas que precisam ser executadas para atingir essa meta, usando um workflow. Isso é capaz de melhorar a visibilidade e a agilidade da lógica de negócio, resultando em representações de nível superior e específicas do domínio, que podem ser compreendidas de maneira mais fácil pelos usuários corporativos de negócio.

3. ESTILO ARQUITETURAL BACKEND TO FRONTEND

Backend To Frontend é um estilo arquitetural que atende as restrições de performance descritas na seção performance e na solução arquitetural.

Atualmente no MCTIC foram definidas duas soluções arquiteturais que implementam o padrão "backend To Frontend": (1) Java Platform Enterprise Edition (JEE); (2) Spring Boot.

Inicialmente, os projetos do MCTIC utilizaram apenas a arquitetura JEE, mas diante de outras necessidades do MCTIC, optou-se por utilizar também o framework Spring Boot.

Dentre as vantagens do Spring boot pode-se destacar o amplo rol de bibliotecas disponíveis para resolver problemas específicos; Facilidade na construção do ambiente das aplicações, porque é apenas um jar; Facilidade na forma de realização de testes de integração e comportamental.

Diante do exposto, o Spring boot também foi adicionado como das opções de solução arquitetural, a ser utilizada dependendo do problema de negócio a ser tratado. Cabe salientar que manter as duas opções de arquitetura oferece a vantagem de encontrar mais profissionais no mercado que possuam conhecimento técnico em algum dos dois frameworks.

4. ARQUITETURA BACKEND TO FRONTEND JEE

Esta seção descreve as camadas da arquitetura, como elas se relacionam entre si e seus papéis para resolver as restrições do MCTIC, tais como: escalabilidade, performance, disponibilidade, confiabilidade e manutenibilidade.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Diagrama de Implantação
-
- Diagrama de Classe
- Interoperabilidade / Escalabilidade / Disponibilidade e Performance
  Manutenibilidade

4.1. Visão geral da arquitetura

4.1.1. Visão de Implantação

A Figura 1.1 ilustra o diagrama de implantação em um "backend to frontend" com dois nós e sua comunicação com sistemas externos, ou seja, apresenta a alocação dos componentes do sistema nos pontos da rede que formam a topologia de hardware onde o sistema será implantado.

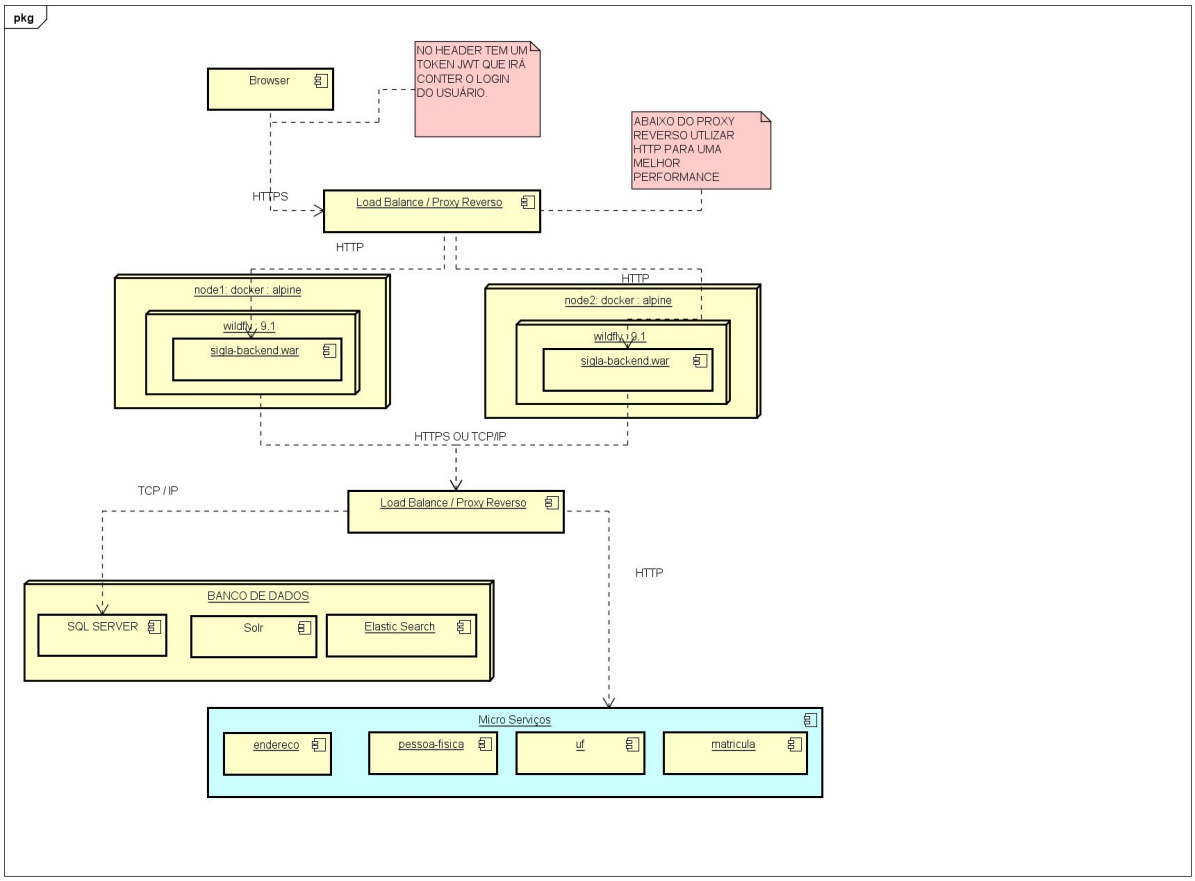


Figura 4 - Visão de implantação JEE

4.1.2. Visão das classes

A Figura 1.2 ilustra o diagrama de classe com os pacotes para cada camada. Os detalhes dos pacotes estão descritos na seção de Manutenibilidade.

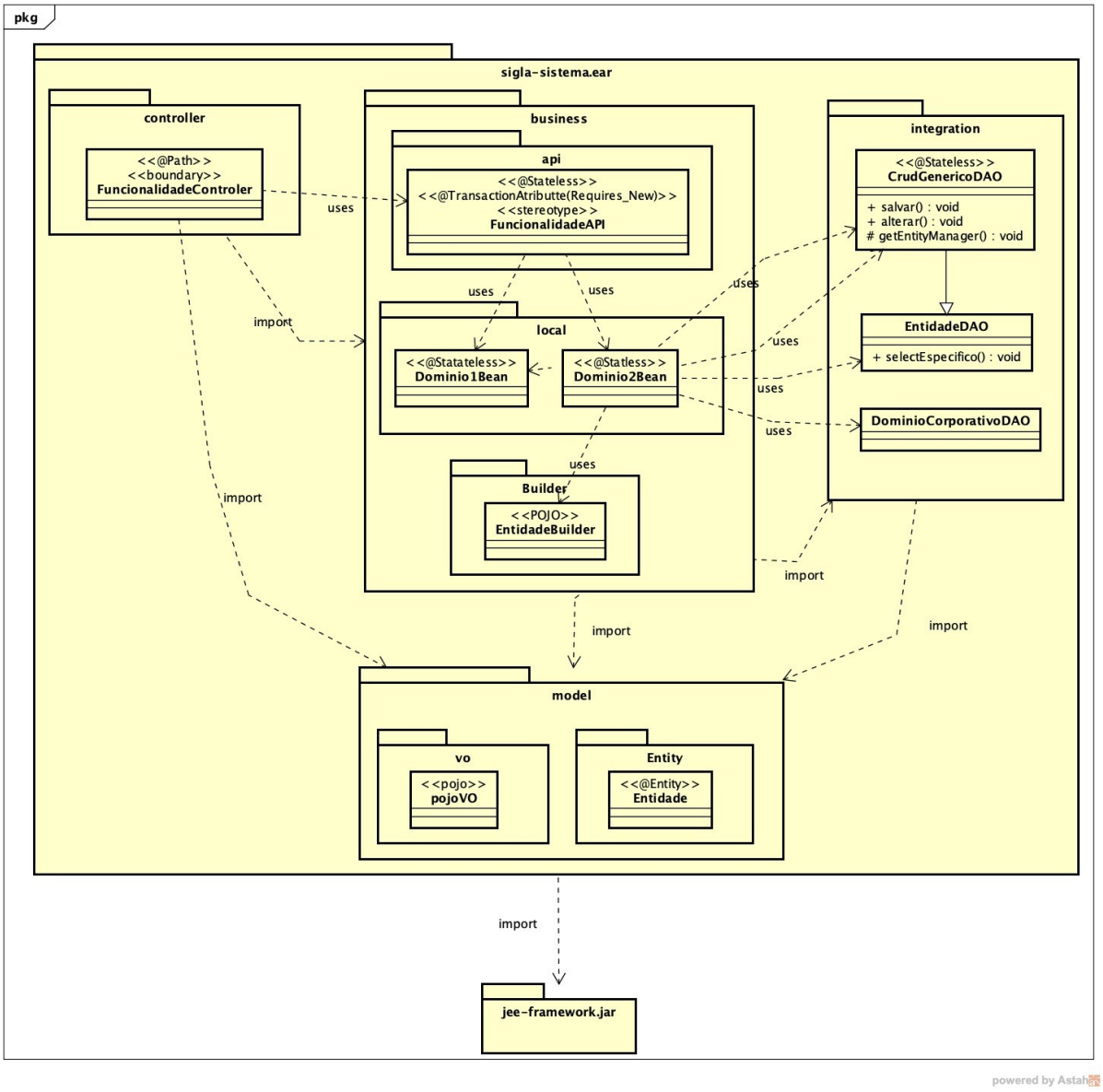


Figura 5 - Visão das Classes JEE

4.2. Requisitos não funcionais atendidos (Restrições)

Esta seção está estruturada através dos requisitos não funcionais de qualidade e a forma como a arquitetura os atende.

4.2.1. Interoperabilidade / Escalabilidade / Disponibilidade e Performance

Para a aplicação ser disponibilizada em ambiente clusterizado utilizando a arquitetura REST e com a performance necessária para satisfazer os requisitos não funcionais elencados na solução arquitetural, são utilizadas as seguintes especificações pertencentes ao padrão (stack) JEE 7.

- JAX-RS: É utilizado para disponibilizar anotações que facilitam a criação de webservices REST;
- EJB Estateless: É utilizado por atender diversas necessidades, como não manter a seção, possuir polling de objetos, possuir controle transacional e proporcionar a injeção de dependências. O motivo de não manter estado o torna bastante performático e simples para escalonamento horizontal, pois não há necessidade de cache distribuído;
- JPA: É utilizado para cache de consultas, o que muitas vezes o torna mais performático do que o JDBC;
- Os serviços rest não irão manter estado, pois o estado será mantido em um token JWT, conforme descrito na solução arquitetural.

4.2.2. Manutenibilidade

Para garantir a manutenibilidade e coesão a aplicação foi dividida em pacotes, conforme descrito na seguinte estrutura de diretórios do projeto:

```
nomeprojeto-backend.war
|
└── br.gov.mctic.sigla
  └──business
  | └── api
  | └── builder
  | └── local
  └── controller
  └── integration
  └── model
```

```
| └── vo
| └── entity
└── util
```

4.2.2.1. Pacote Model

Os "Objetos anêmicos" são objetos que possuem apenas atributos e relacionamentos com outros objetos e não possuem comportamento. Em outras palavras, não possuem regra de negócio nos métodos. Eles são utilizados na arquitetura no pacote model, sendo que esse pacote poderá conter apenas dois tipos de objetos:

- @Entity: Objetos que representam tabelas e relacionamentos no banco de dados.
- VOS: Objetos que compõem várias entidades ou outros valores a serem trafegados para as camadas da aplicação.

Importante: preferencialmente deve-se usar entidades como parâmetros e retorno de classes "APIs" e "Controllers".

O padrão arquitetural utiliza "serviços ricos", que consiste ter as regras de negócio na camada de serviços (pacote business) e trafegar objetos anêmicos entre as camadas, como no exemplo abaixo:

- O Objeto irá trafegar pelas seguintes camadas/pacotes:
    - (REST) -> controller -> business -> integration

Regras	**Motivo**
Não adicionar comportamento (regra de negócio) nas entidades	O padrão arquitetural utiliza "serviços ricos"

4.2.2.2. Pacote Controller

Neste pacote devem ser implementados os web services REST que fazem delegações para a camada business.

Regras	**Motivo**
Devem chamar as regras de negócio utilizado apenas classes do pacote "api".	As classes do pacote API são as boundarys da regra de negócio do sistema. Ela inicia uma transação o que facilita saber quais regras de negócio são expostas de forma externa melhorando a coesão do sistema fazendo que, caso haja manutenção da camada de negócio, não afete a camada de controller
Todas as classes devem conter o sufixo "Controller"	Como a arquitetura utiliza o padrão MVC é uma boa prática colocar o nome do padrão no elemento.

4.2.2.3. Pacote Business

No pacote business devem ser colocadas todas as regras de negócio e "builders".

4.2.2.4. Pacote Business/API

Regras	**Motivo**
Devem conter o sufixo "API"	Para facilitar as pesquisas via IDE as classes API.
Devem utilizar as anotações @Stateless @TransactionAtributte(Requires_New)	A classe API é boundary do sistema, por esse motivo, a transação deve sempre ser iniciada nessa classe.

Classes APIS podem utilizar classes do pacote "integration". Ex DAO	Agilizar o desenvolvimento, evitando utilizar a classe API apenas para fazer delegações
Classes API só podem ser utilizadas pelas controller	As classes API são classes boundarys que serão serviços que serão expostos para outros sistemas. Elas sendo chamadas apenas pela controller dá para saber aonde começou a transação, facilitando a manutenção do sistema
Trafegue entidades (@Entity) ou VOs, de preferência a entidades para enviar conversões	O padrão arquitetural do JEE utiliza serviços ricos no qual consiste ter serviços na camada de negócio e trafegar objetos anêmicos

4.2.2.5. Pacote Integration

Deve contemplar todo código que irá fazer acesso a sistemas externos.

Regras	**Motivo**
Utilizar SQL nativo apenas em consultas complexas	O ORM facilita a manutenção quando é trocado o nome de tabelas e colunas no sistema.
Acesso a outros sistemas devem ser inseridos no pacote integration	Caso mude a tecnologia dos outros sistemas irá afetar apenas a camada de integração.
Não utilizar funções específicas do banco de dados	Caso mude o banco de dados a aplicação irá gerar grande refatoração.

4.2.2.6. Útil

Utilitários POJOS que podem ser utilizados em todas camadas, por exemplo: formatação de texto.

4.2.3. Flexibilidade

O código java compilado executa utilizando a JVM, existem JVM para diversos sistemas operacionais o que torna o java flexível.

5. ARQUITETURA BACKEND TO FRONTEND SPRING BOOT

Esta seção descreve as camadas da arquitetura, como elas se relacionam entre si e seus papéis para resolver as restrições do MCTIC, tais como: escalabilidade, performance, disponibilidade, confiabilidade e manutenibilidade.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Diagrama de Implantação;
- Diagrama de Classe;
- Interoperabilidade / Escalabilidade / Disponibilidade e Performance Manutenibilidade.

5.1. Visão geral da arquitetura

5.1.1. Visão de Implantação

A Figura 1.1 ilustra o diagrama de implantação em um "backend to frontend" com dois nós e sua comunicação com sistemas externos, ou seja, apresenta a alocação dos componentes do sistema nos pontos da rede que formam a topologia de hardware onde o sistema será implantado.

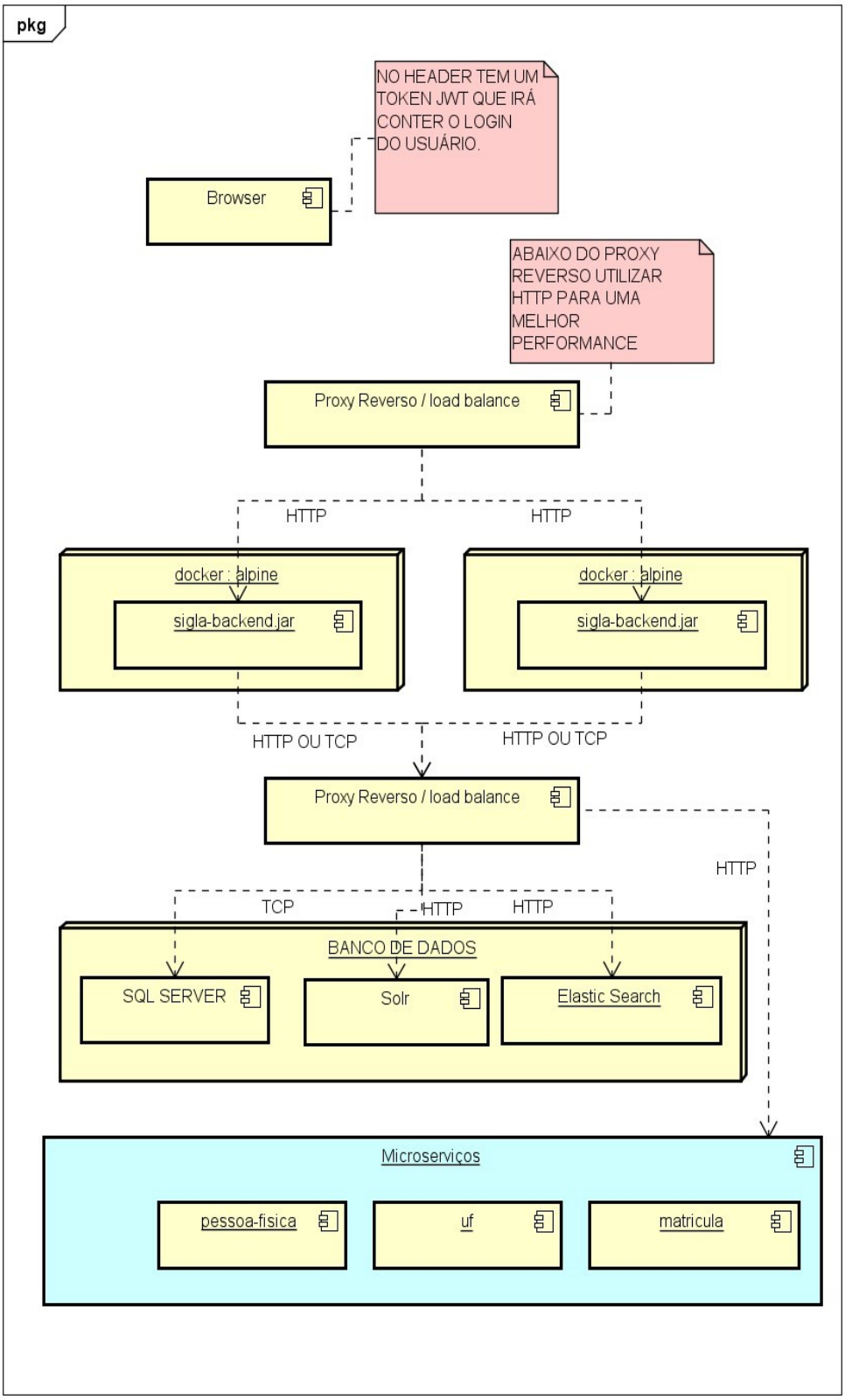


Figura 6 - Visão de implantação Spring Boot

5.1.2. Visão das classes

A Figura 1.2 ilustra o diagrama de classe com os pacotes para cada camada. Os detalhes dos pacotes estão descritos na seção de Manutenibilidade.

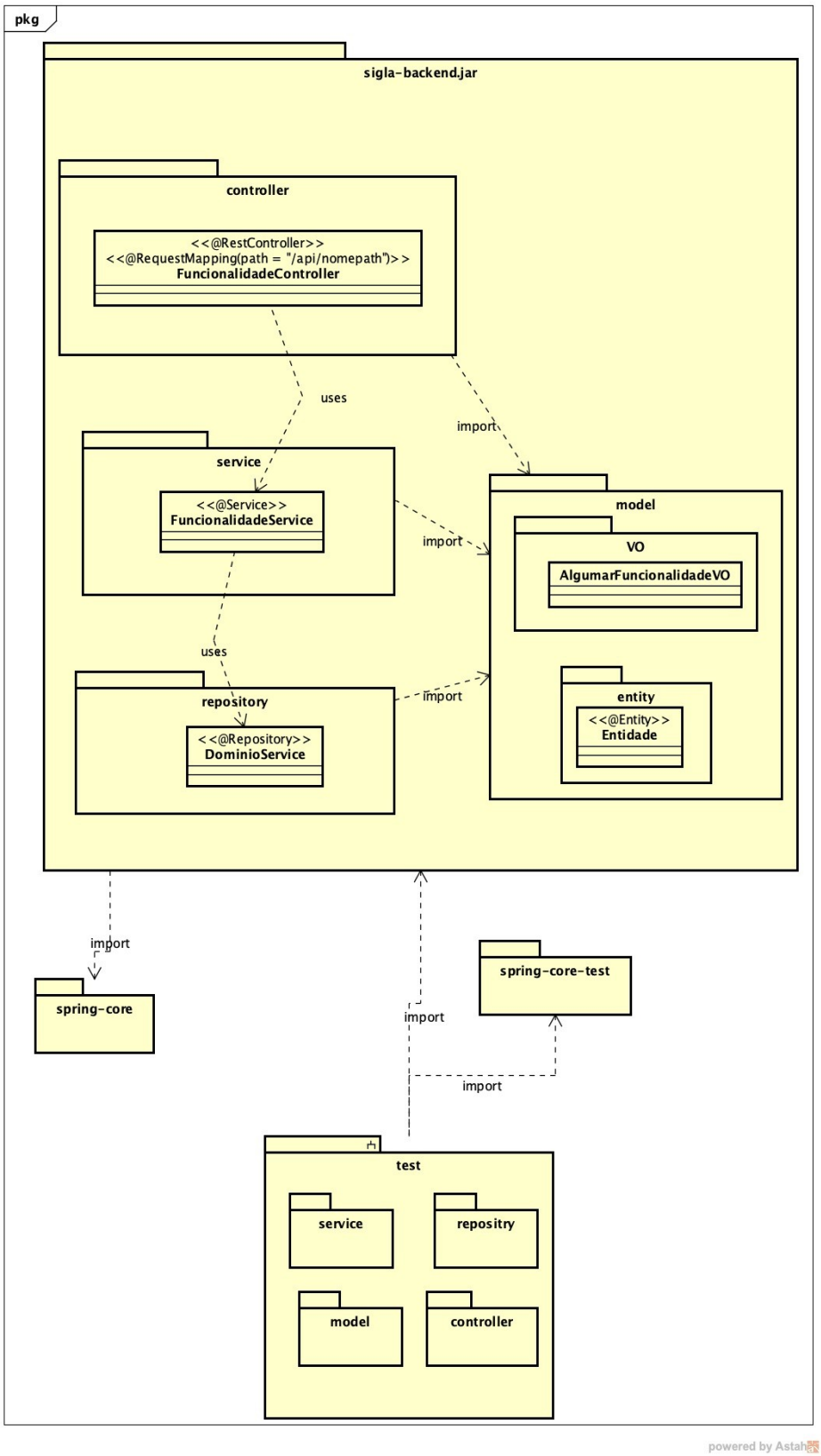


Figura 7 - Visão das Classes Spring Boot

5.2. Requisitos não funcionais atendidos (Restrições)

Esta seção está estruturada através dos requisitos não funcionais de qualidade e a forma como a arquitetura os atende.

5.2.1. Interoperabilidade / Escalabilidade / Disponibilidade e Performance

Para a aplicação ser disponibilizada em ambiente clusterizado utilizando a arquitetura REST e com performance necessária para satisfazer os requisitos não funcionais elencados na solução arquitetural, são utilizadas as seguintes especificações pertencentes ao framework Spring Boot.

- @Service: É utilizado, pois possui anotações para uma fácil criação de web service REST;
- @Repositor: É utilizado, pois não mantém seção e implementa o padrão singleton também possui outras características, tais como: controle transacional e injeção de dependências. O motivo de não manter estado o torna bastante performático e simples para escalonamento horizontal, pois não há necessidade de cache distribuído.
- SpringData é utilizado pois possui cache de consultas o que muitas vezes o torna mais performático do que o JDBC, e também possui integração com banco de dados orientados a documento (NoSql).
- Os serviços rest não irão manter estado, pois o estado será mantido em um token JWT, conforme descrito na solução arquitetural.

5.2.2. Manutenibilidade

Para garantir a manutenibilidade e coesão a aplicação foi dividida em pacotes, conforme descrito na seguinte estrutura de diretórios do projeto:

```
src
└──
|  main / java
|  |
|  └── br.gov.mctic.sigla
|     └── controller
|     └── service
|     └── repository
|     └── model
|     |  └── vo
|     |  └── entity
|     └── configuration
|
|
└──
|  test/java
|  |
|  └── br.gov.mctic.sigla
|     └── controller
|     └── service
|     └── repository
|     └── model
```

5.2.2.1. Pacote Controller

Neste pacote devem ser implementados os web services REST que fazem delegações para a camada business.

Regras	Motivo
Todas as classes devem conter o sufixo "Controller"	Como a arquitetura utiliza o padrão MVC é uma boa prática colocar o nome do padrão no elemento.

5.2.2.2. Pacote service

No pacote service devem ser colocadas todas as regras de negócio.

Regras	Motivo
Devem conter o sufixo Service	Para facilitar as pesquisas via IDE as classes Service é também um padrão do Spring Boot.
Devem utilizar a anotação @Service	o motivo está elencado nos requisitos não funcionais

5.2.2.3. Pacote Repository

Deve ser colocado todo código que irá fazer acesso a sistemas externos.

Regras	Motivo
Utilizar SQL nativo apenas em consultas complexas	O ORM facilita a manutenção quando é trocado o nome de tabelas e colunas no sistema.
Acesso a outros sistemas devem ser colocados no pacote integration	Caso mude a tecnologia dos outros sistemas irá afetar apenas a camada de integração
Não utilizar funções específicas do banco de dados	Caso mude o banco de dados a aplicação irá gerar grande refatoração.

5.2.2.4. Pacote Model

Os "Objetos anêmicos" são objetos que possuem apenas atributos e relacionamentos com outros objetos e não possuem comportamento. Em outras palavras, não possuem regra de negócio nos métodos. Eles são utilizados na arquitetura no pacote model, sendo que esse pacote poderá conter apenas dois tipos de objetos:

- @Entity: Objetos que representam tabelas e relacionamentos no banco de dados.
- VOS: Objetos que compõem várias entidades ou outros valores a serem trafegados para as camadas da aplicação.

Importante: dê preferência a usar entidades como parâmetros e retorno de classes "APIs" e "Controller

Regras	**Motivo**
Não adicionar comportamento (regra de negócio) nas entidades	O padrão arquitetural utiliza serviços ricos no qual consiste ter serviços na camada de negócio e trafegar objetos anêmicos

5.2.2.5. Pacote Configuration

Classes para a configuração do projeto.

5.2.2.6. Pacote Tests

Classes que irão realizar os testes de integração, unitários e comportamentais do sistema.

5.2.3. Flexibilidade

O código java compilado é executado utilizando a JVM, existem JVM para diversos sistemas operacionais o que torna o java flexível, pois é compatível em diversos tipos de sistemas operacionais.

6. ARQUITETURA FRONTEND ANGULAR JS

Esta seção descreve as camadas da arquitetura, como elas se relacionam entre si e seus papéis para resolver as restrições do MCTIC, tais como: escalabilidade, performance, disponibilidade, confiabilidade e Manutenibilidade.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Diagrama de implantação
- Diagrama de classe
- Requisitos não funcionais atendidos
- Interoperabilidade / escalabilidade / disponibilidade e performance

6.1. Visão geral da arquitetura

6.1.1. Visão de Implantação

A Figura 1.1 ilustra o diagrama de implantação em um "backend to frontend" com dois nós, ou seja, apresenta a alocação dos componentes do sistema nos pontos da rede que formam a topologia de hardware onde o sistema será implantado.

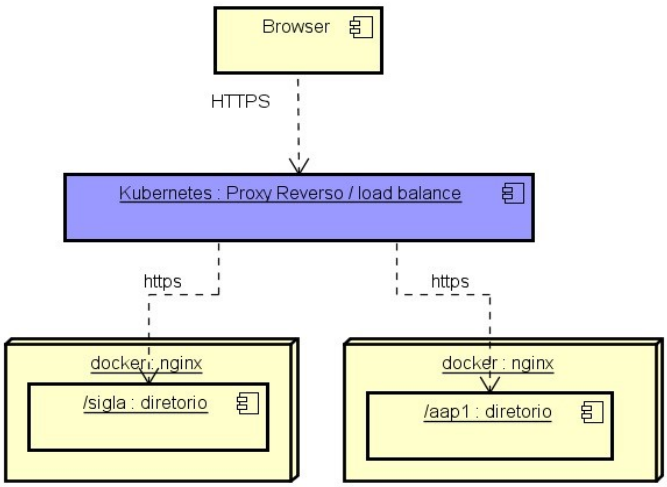


Figura 8 - Visão de implantação Angular JS

6.1.2. Visão das classes

A Figura 1.2 ilustra o diagrama de classe com os pacotes para cada camada. Os detalhes dos pacotes estão descritos na seção de Manutenibilidade.

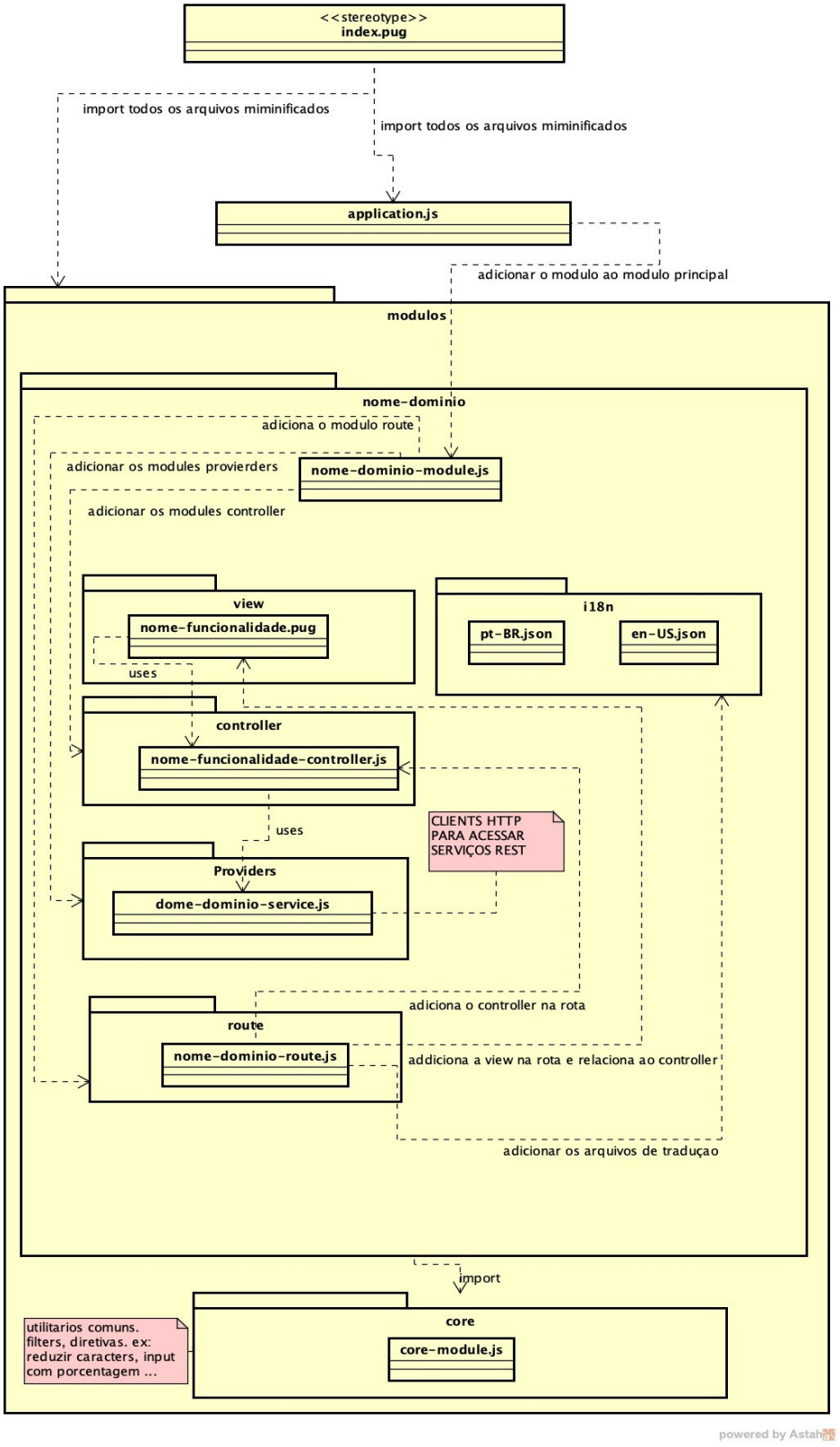


Figura 9 - Visão das Classes Angular JS

6.2. Requisitos não funcionais atendidos

Esta seção está estruturada através dos requisitos não funcionais de qualidade e a forma como a arquitetura os atende.

6.2.1. Interoperabilidade / Escalabilidade / Disponibilidade e Performance

As páginas são estáticas e hospedadas em um cluster com um Load Balance, onde o token JWT será armazenado no SessionStorage do browser.

6.2.2. Manutenibilidade

Para melhor manutenibilidade a estrutura de diretórios é dividida em módulos, conforme a orientação do "style guide" de John Papa, referência na disciplina. Nesse guia é possível obter boas práticas que resultam em uma melhor manutenibilidade.

A Estrutura de pastas da aplicação e detalhada nessa seção Estrutura de pastas

Links do style guide do John Papa:

- Github (recomendado);
- Local (caso esteja sem internet).

6.2.3. Usabilidade

No que tange usabilidade, pode-se destacar os seguintes requisitos não funcionais:

- Os sistemas devem ser responsivos para serem acessados por vários tipos de dispositivos via WEB.

Para atender esse requisito foi escolhido utilizar a especificação de interfaces MATERIAL.IO, que foi criada pela Google, e possui diversas regras que visam atender os requisitos de usabilidade e responsividade para diversos tipos de dispositivos.

O framework escolhido para o desenvolvimento de Interfaces web foi o AngularJS por utilizar a arquitetura Single Page Application (SPA). O framework de componentes escolhido foi o MaterialJS, pois ele implementa a especificação do MATERIAL.IO.

7. ARQUITETURA DE MICROSERVIÇO SPRING BOOT

Esta seção descreve as camadas da arquitetura, como elas se relacionam entre si e seus papéis para resolver as restrições do MCTIC, tais como: escalabilidade, performance, disponibilidade, confiabilidade e Manutenibilidade.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Diagrama de Implantação
- Diagrama de Classe
- Interoperabilidade / Escalabilidade / Disponibilidade e Performance
  Manutenibilidade

7.1. Visão geral da arquitetura

7.1.1. Visão de Implantação

A Figura 1.1 ilustra o diagrama de implantação em um "backend to frontend" com dois nós e sua comunicação com sistemas externos, ou seja, apresenta a alocação dos componentes do sistema nos pontos da rede que formam a topologia de hardware onde o sistema será implantado.

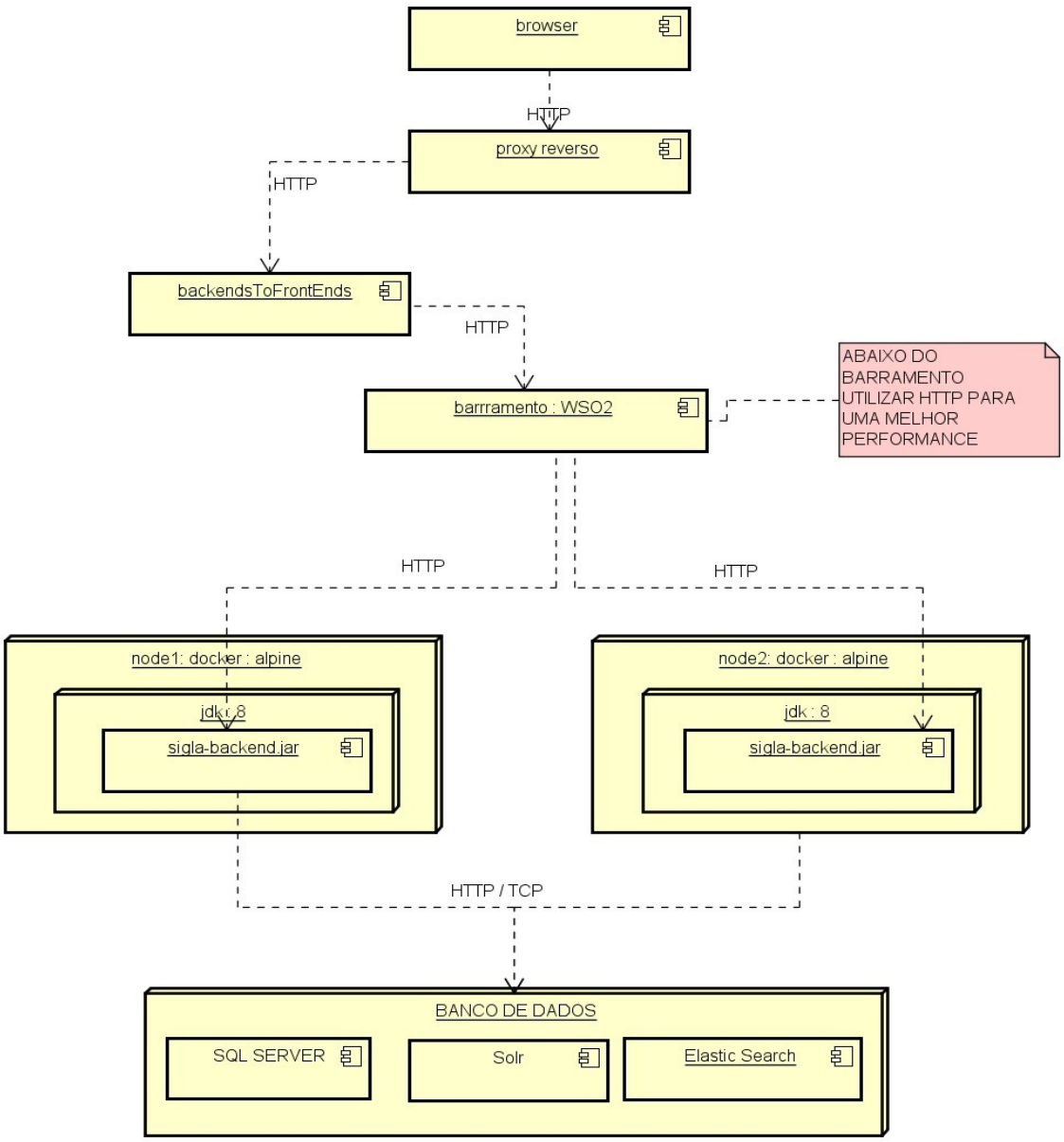
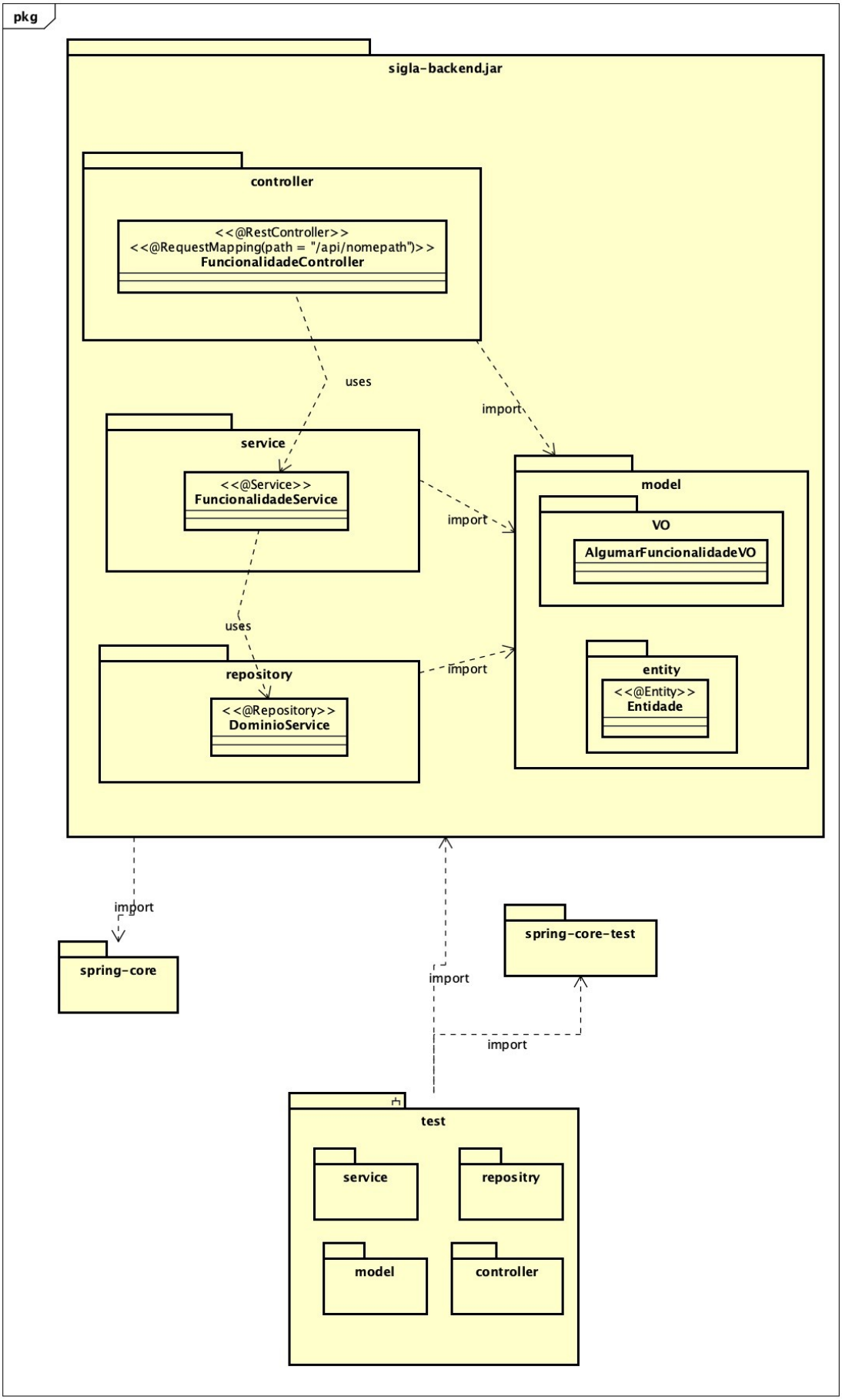


Figura 10 - Visão de implantação microserviço Spring Boot

7.1.2. Visão das classes

A Figura 1.2 ilustra o diagrama de classe com os pacotes para cada camada. Os detalhes dos pacotes estão descritos na seção de Manutenibilidade.



powered by Astah

Figura 11 - Visão das Classes microserviço Spring Boot

7.2. Requisitos não funcionais atendidos (Restrições)

Esta seção está estruturada através dos requisitos não funcionais de qualidade e a forma como a arquitetura os atende.

7.2.1. Interoperabilidade / Escalabilidade / Disponibilidade e Performance

Para a aplicação ser disponibilizada em ambiente clusterizado utilizando a arquitetura REST e com a performance necessária para satisfazer os requisitos não funcionais elencados na solução arquitetural, são utilizadas as seguintes especificações pertencentes ao padrão (stack) Spring Boot.

- @Service: É utilizado, pois possui anotações para uma fácil criação de web service REST.
- @Repository: É utilizado, pois não mantém seção e implementa o padrão singleton também possui outras características, tais como: controle transacional e injeção de dependências. O motivo de não manter estado o torna bastante performático e simples para escalonamento horizontal, pois não há necessidade de cache distribuído.
- SpringData é utilizado pois possui cache de consultas o que muitas vezes o torna mais performático do que o JDBC, e também possui integração com banco de dados orientados a documento (NoSql).
- Os serviços rest não irão manter estado, pois o estado será mantido em um token JWT, conforme descrito na solução arquitetural

7.2.2. Manutenibilidade

Para garantir a manutenibilidade e coesão a aplicação foi dividida em pacotes, conforme descrito na seguinte estrutura de diretórios do projeto:

```
src
└──
|  main / java
|  |
|  └── br.gov.mctic.sigla
|     └── controller
|     └── service
|     └── repository
|     └── model
|     |  └── vo
|     |  └── entity
|     └── configuration
|
|
└──
|  test/java
|  |
|  └── br.mctic.gov.sigla
|     └── controller
|     └── service
|     └── repository
|     └── model
```

7.2.2.1. Pacote Controller

Neste pacote devem ser implementados os web services REST que fazem delegações para a camada business.

Regras	Motivo
Todas as classes devem conter o sufixo "Controller"	Como a arquitetura utiliza o padrão MVC é uma boa prática colocar o nome do padrão no elemento.

7.2.2.2. Pacote service

No pacote service devem ser colocadas todas as regras de negócio.

Regras	Motivo
Devem conter o sufixo Service	Para facilitar as pesquisas via IDE as classes Service é também um padrão do Spring Boot.
Devem utilizar a anotação @Service	o motivo está elencado nos requisitos não funcionais

7.2.2.3. Pacote Repository

Deve ser colocado todo código que irá fazer acesso a sistemas externos.

Regras	Motivo
Utilizar SQL nativo apenas em consultas complexas	O ORM facilita a manutenção quando é trocado o nome de tabelas e colunas no sistema.
Acesso a outros sistemas devem ser colocados no pacote integration	Caso mude a tecnologia dos outros sistemas irá afetar apenas a camada de integração
Não utilizar funções específicas do banco de dados	Caso mude o banco de dados a aplicação irá gerar grande refatoração.

7.2.2.4. Pacote Model

Os "Objetos anêmicos" são objetos que possuem apenas atributos e relacionamentos com outros objetos e não possuem comportamento. Em outras palavras, não possuem regra de negócio nos métodos. Eles são utilizados na arquitetura no pacote model, sendo que esse pacote poderá conter apenas dois tipos de objetos:

- @Entity: Objetos que representam tabelas e relacionamentos no banco de dados.
- VOS: Objetos que compõem várias entidades ou outros valores a serem trafegados para as camadas da aplicação.

Importante: dê preferência a usar entidades como parâmetros e retorno de classes "APIs" e "Controller

Regras	**Motivo**
Não adicionar comportamento (regra de negócio) nas entidades	O padrão arquitetural utiliza serviços ricos no qual consiste ter serviços na camada de negócio e trafegar objetos anêmicos

7.2.2.5. Pacote Configuration

Classes para a configuração do projeto.

7.2.2.6. Pacote Tests

Classes que irão realizar os testes de integração, unitários e comportamentais do sistema.

7.2.3. Flexibilidade

O código java compilado é executado utilizando a JVM, existem JVM para diversos sistemas operacionais o que torna o java flexível, pois é compatível em diversos tipos de sistemas operacionais.

8. ARQUITETURA DE MICROSERVIÇO MOLECULAR JS (NODE)

Esta seção descreve as camadas da arquitetura, como elas se relacionam entre si e seus papéis para resolver as restrições do MCTIC, tais como: escalabilidade, performance, disponibilidade, confiabilidade e Manutenibilidade.

O restante dos tópicos desta seção está estruturado da seguinte forma:

- Diagrama de Implantação
- Diagrama de Classe
- Interoperabilidade / escalabilidade / disponibilidade e performance Manutenibilidade

8.1. Visão geral da arquitetura

8.1.1. Visão de Implantação

A Figura 1.1 ilustra o diagrama de implantação em um "backend to frontend" com dois nós e sua comunicação com sistemas externos, ou seja, apresenta a alocação dos componentes do sistema nos pontos da rede que formam a topologia de hardware onde o sistema será implantado

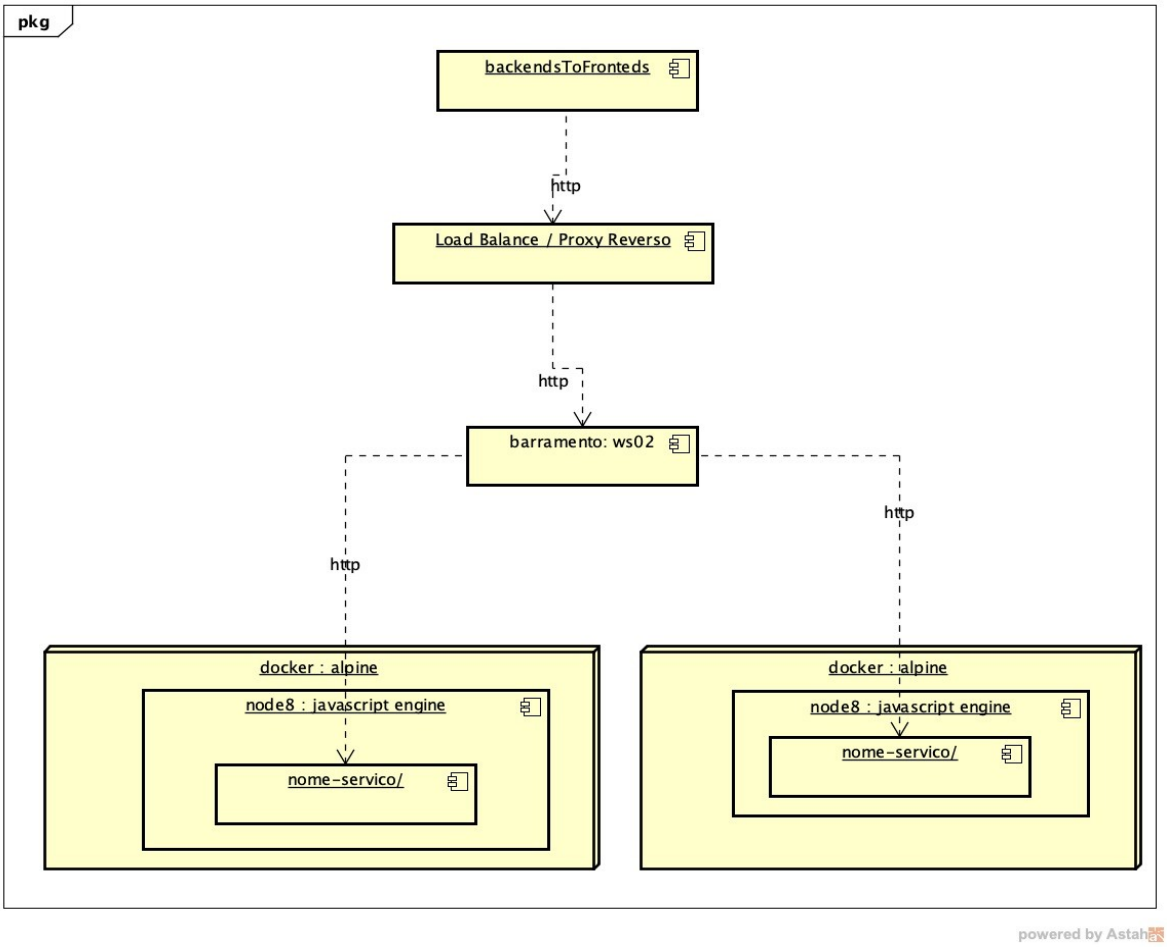


Figura 12 - Visão de implantação microserviço Molecular JS (NODE)

8.1.2. Visão das classes

A Figura 1.2 ilustra o diagrama de classe com os pacotes para cada camada. Os detalhes dos pacotes estão descritos na seção de Manutenibilidade.

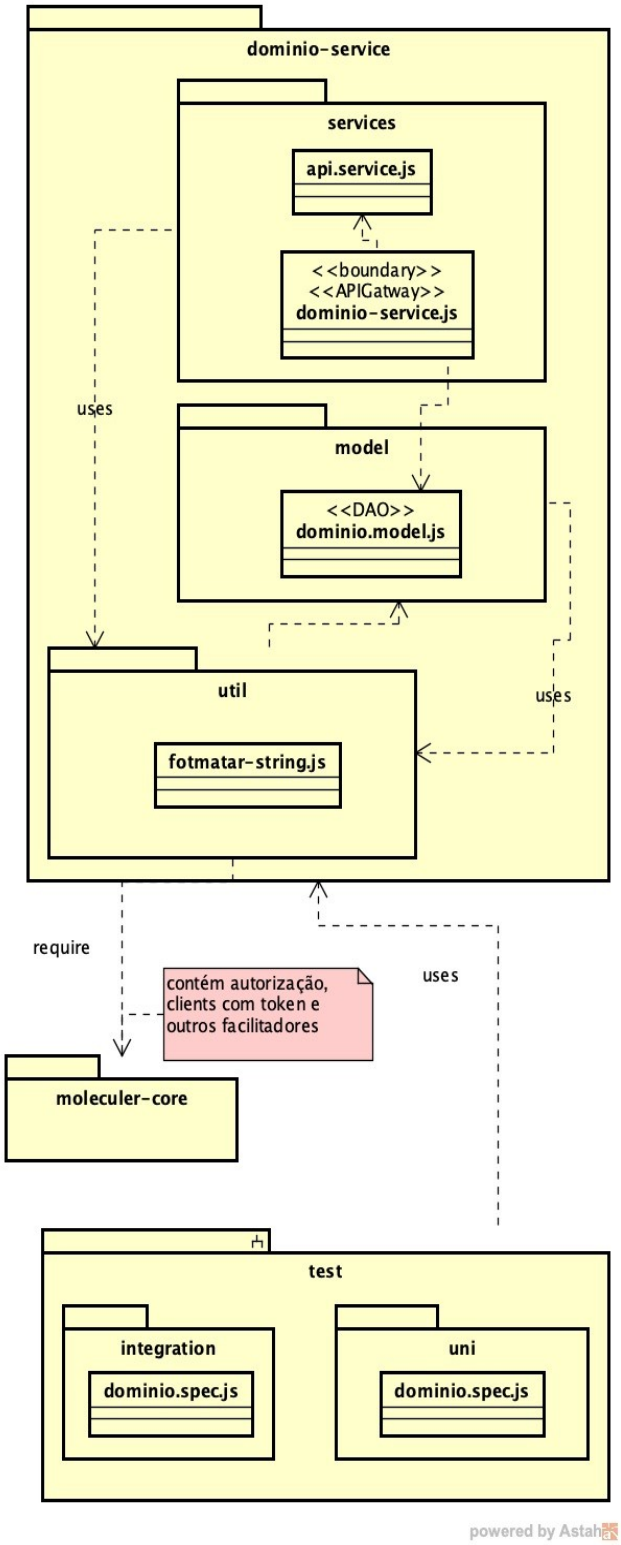


Figura 11 - Visão das Classes microserviço Molecular JS (NODE)

8.2. Requisitos não funcionais atendidos (Restrições)

Esta seção está estruturada através dos requisitos não funcionais de qualidade e a forma como a arquitetura os atende.

8.2.1. Interoperabilidade / Escalabilidade / Disponibilidade e Performance

Os microserviços proveem serviços REST, o que possibilita a interoperabilidade para prover serviços para outras aplicações.

Os microserviços são implantados em ambientes clusterizados com load balance, o que permite escalonamento horizontal, que irá atender os requisitos de escalabilidade, disponibilidade e performance.

O framework MoleculerJS possui cache de serviço REST que pode ser facilmente configurado o tempo, o que aumenta significativamente a performance.

8.2.2. Manutenibilidade

Um microserviço possui poucas funcionalidades, o que facilita a manutenibilidade, porque o serviço pode ser reescrito totalmente em um curto prazo de tempo (tempo médio 1 semana).

8.2.3. Flexibilidade

O NODE é uma engine que processa java script e transforma em linguagem de máquina, grande parte do seu core é escrito em c++. O NODE pode ser instalado em diversos sistemas operacionais. O que garante uma flexibilidade.

9. REFERÊNCIAS

- ANGULAR.IO - http://angular.io/
- ANGULAR.JS - https://angularjs.org/
- ANGULAR.JS - https://material.angularjs.org
- Angular Style Guide - https://github.com/johnpapa/angular-styleguide
- KEYCLOAK - https://www.keycloak.org/
- SOLR - http://lucene.apache.org/solr/
- ELASTIC SEARCH- https://www.elastic.co/
- JBPM - https://www.jbpm.org/
- NODE JS - https://moleculer.services/
- SPRING BOOT - http://spring.io/projects/spring-boot