Ministério da Justiça
Coordenação Geral de Tecnologia da Informação
MJ

Projeto: Arquitetura de Referência Java e Angular
Cliente: MJ

Modelo de Arquitetura

Autor: Diogo Carvalho de Matos

Brasília-DF
23 de Julho de 2019

1 Sumário

2 Dados Gerais

Projeto/Sistema: Arquitetura de Referência

3 Finalidade

Os artefatos da Arquitetura de Referência fazem parte da base de ativos reutilizáveis de uma organização. A finalidade desses artefatos é criar um ponto de partida para o desenvolvimento da arquitetura. Eles podem variar de padrões de arquitetura, mecanismos de arquitetura e frameworks prontos a sistemas completos, com características conhecidas e de uso comprovado. Eles podem ser aplicados de forma geral ou para uma ampla classe de sistemas de domínios ou ter um enfoque mais limitado, específico de domínio.

O uso de arquiteturas de referência testadas é uma maneira eficaz de lidar com os vários requisitos não-funcionais (particularmente os requisitos de qualidade), selecionando arquiteturas de referência existentes, que são conhecidas através do uso para atender a esses requisitos. As arquiteturas de referência podem existir ou ser usadas em diferentes níveis de abstração e a partir de diversos pontos de vista. Esses pontos de vista correspondem às Visões 4+1. Desse modo, o arquiteto de software pode selecionar a visão que seja mais adequada — apenas um design de arquitetura ou um design e uma implementação, em variados graus de conclusão.

4 Modelo de Arquitetura

4.1 **Objetivos do Documento**

O objetivo deste documento é explicitar o modelo arquitetural utilizado no desenvolvimento dos sistemas do Ministério da Justiça. São também descritos neste documento a descrição dos focos e sistemáticas arquiteturais, descrição das camadas quem compõem o modelo arquitetural, e requisitos de segurança desempenho e integração. Este documento também tem o objetivo de orientar todo o pessoal técnico envolvido nas equipes de desenvolvimento, oferecendo diretrizes quanto às tecnologias e padrões a serem utilizadas no projeto.

4.2 **Metas e Restrições da Arquitetura**

A meta da arquitetura é suportar os requisitos funcionais, ou de negócio, e os requisitos não funcionais da aplicação como: Manutenabilidade, Interoperabilidade, Portabilidade, Reusabilidade, Confiabilidade, Integridade e Segurança.

A arquitetura definida é baseada na “stack” Java EE 8 que permite que seja adotada uma abordagem corporativa para o desenvolvimento de software em larga escala, através da implementação de componentes de negócio, serviços e componentes de interface gráfica reutilizáveis de forma desacoplada viabilizando a produtividade e padronização no desenvolvimento.

Com a arquitetura de referência definida aqui neste documento é possível adotar a estratégia de implementação segregada, server side e client side, ou seja, entre componentes de negócio e componentes de interface gráfica, onde, são desacoplados e se comunicam através de um protocolo interoperável.

4.3 Público Alvo

Analistas de sistemas, desenhistas de software, programadores, gerentes de projeto e demais interessados.

4.4 Definições, Acrônimos e Abreviações

- **JDK** - Java Development Kit.
- **Java EE** - Java Enterprise Edition.
- **SPA** - Single Page Applications.
- **MVC** - Model-view-controller.
- **HTML** - HyperText Markup Language.
- **CSS** - Cascading Style Sheets.
- **DOM** - Document Object Mode.
- **OOP** - Object-oriented programming.
- **UI** - User Interface Design.
- **EAR** - Enterprise Application aRchive.
- **API** - Application Programming Interface.
- **DTO** - Data Transfer Object.
- **DAO** - Data Access Object.
- **JSON** - JavaScript Object Notation.
- **XML** - Extensible Markup Language.
- **HTTP** - Hypertext Transfer Protocol.
- **VO** - Value Object.
- **EJB** - Enterprise Java Bean.
- **BO** - Business Object.
- **JMS** - Java Message Service.
- **IDE** - Integrated Development Environment.
- **CLI** - Command-line interface.

4.5 Arquitetura de Referência e Padrões de Arquitetura

O projeto é desenvolvido utilizando o modelo arquitetural ***cliente-servidor***, dividido em três camadas distintas: o *frontend, desenvolvido em Typescript* e projetado com o *framework Angular;* uma API (REST) desenvolvida em Java 12 (JDK 12u1), e projetado seguindo as especificações *Java EE 8* e um *client* para a integração com projetos terceiros.

1) **Frontend**: É uma *Single Page Application* (SPA) em *Angular*, que é um framework *MVC* baseado em componentes. O *Angular* é totalmente baseado em componentes (*Component-based*), onde, um componente encapsula uma estrutura (HTML), estilo (CSS) e comportamento (*JavaScript*) e podem ser utilizadas como *tags* HTML customizadas, o que resulta em um grande reaproveitamento de código fonte e padronização durante o processo de desenvolvimento dos projetos. Além dos componentes o *Angular* oferece uma vastas opções de diretivas, além de permitir que o desenvolvedor crie suas próprias diretivas customizadas, as diretivas atuam manipulando diretamente o DOM é o recurso correto quando esse comportamento for necessário. Uma outra característica do *Angular* é o uso de *TypeScript*, uma das principais vantagens do uso dessa linguagem é que permite o desenvolvedor escrever código J*avaScript* com muitos conceitos de *OOP*, muito mais organizado e fácil de manter, sendo possível utilizar lambda, *generics*, heranças e interfaces. Apesar da aplicação ser escrita em *TypeScript* quando a aplicação entrar no processo de compilação será executado um recurso denominado *"Transpile"* que transformará todo o código para o *JavaScript* que é a linguagem interpretada no browser. Para os componentes visuais é o utilizado o framework *Angular Material*, este framework é mantido pelo Google e é baseado na especificação Material Design.

2) **Backend**: A estrutura da aplicação deve utilizar o padrão *Layer*. É criado um sistema multimódulos, onde, cada projeto representa uma camada que são empacotados em um *EAR*. Abaixo, será descrito como são divididos estes módulos e a responsabilidade de cada um deles.

    a) **modulo-rest**: Módulo que expõe *APIs (WebServices)* a serem acessadas por uma interface de usuário desacoplada e/ou aplicações que desejam consumir algum "serviço" do projeto. Além de fornecer uma API clara, esta camada é também um bom local para aplicar segurança. Normalmente a implementação é feita utilizando-a como uma *facade* ou *gateway*, de modo que todo o comportamento real estará nos objetos de negócio. Neste caso, é fornecida uma *API* mais fácil de usar porque, tipicamente ela é orientada a casos do uso. Esta camada é apropriada para cenários de chamadas remotas, por perspectiva da granularidade da interface que a mesma implementa. Mas essa situação tem o custo de lidar com distribuição desses objetos. Trabalho esse como implementação de assinatura dos métodos, aplicabilidade de uma tecnologia de aplicação distribuída, objetos de transferência de dados (*DTO*), *tracing,*

exception handler e etc. Seus benefícios são bem consideráveis, agregando maior abstração do consumo dos componentes de negócio, fornecendo recurso para distribuição da aplicação, definição clara das operações disponíveis de uma aplicação, ponto centralizado para implementação da autenticação e autorização para o consumo de uma regra de negócio.

b) **modulo-service**: Módulo responsável pelas regras de negócio da aplicação (*Business project*). É nele que ficam as funções e regras de todo o negócio. Não existe uma interface para o usuário e seus dados são voláteis, ou seja, para que algum dado seja mantido deve ser utilizada a camada de dados. A camada de negócio é responsável por controlar todo o fluxo da informação e o controle transacional. O componente de negócios encapsula a lógica de negócios em um nível intermediário compartilhado, ou seja, todos os clientes acessam os mesmos objetos de negócio, que estão em uma localização centralizada. Isto facilita a redundância e a manutenção das aplicações clientes. A camada de negócios possibilita o processamento distribuído dos dados, distribuindo o serviço da aplicação entre diversas máquinas, podendo melhorar o desempenho graças a equalização de carga, onde as requisições podem ser gerenciadas pelos diversos servidores de aplicação existentes e ainda permite que um dado servidor de aplicação saia do ar, sem prejudicar o funcionamento do sistema, pois o servidor de aplicação vai estar replicado em outras máquinas servidoras. Com a camada de negócios é possível isolar funcionalidades sensíveis em camadas com diferentes restrições de acesso. Isto provê níveis de segurança mais flexíveis e configuráveis. As camadas intermediárias podem limitar os pontos de entrada para determinadas funcionalidades, facilitando o controle.

c) **modulo-persistence**: Módulo responsável pela persistência da aplicação (*DAO*). Esta camada de dados é acessada pela camada de negócio para ler e escrever dados. Os dados podem vir de diversas origens como: bancos de dados relacionais, bancos de dados *NoSQL*, sistemas de arquivos distribuídos, componentes de negócio e web services de terceiros.

4.6 Padrões de Desenho

Os padrões guiam a arquitetura do sistema e permitem o entendimento global em termos dos componentes que o formam e como estes interagem entre si. Abaixo estão listados os padrões mais recorrentes à aplicação, que guiarão a arquitetura:

O projeto será composto por uma arquitetura cliente-servidor, onde o cliente (client-side) será responsável pela UI, inputs, validação de formulários, lógica e estado, enquanto o servidor (server-side) será responsável pela lógica de negócio, modelo de domínio e persistência das informações. A comunicação entre as duas partes é realizada através do protocolo WEB/HTTP, sendo JSON e XML os formatos mais comuns para a troca de informações.

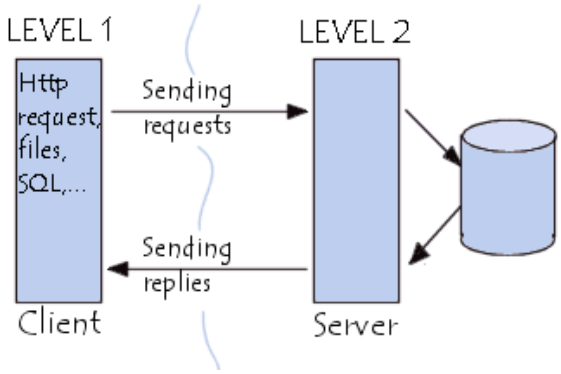

Figura 1: Cliente-Servidor

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

- **Model**: Essa camada consiste na manipulação dos dados, sendo responsável pela escrita e leitura das informações e também de suas validações (business layer);
- **Controller**: Essa camada atua como uma interface entre a View e o Model;
- **View**: Essa camada é responsável pela interação com o usuário final, servirá apenas para exibir informações enviadas pelo controller, nessa camada não deve existir nenhuma lógica ou regra de negócio, apenas regras de interface.

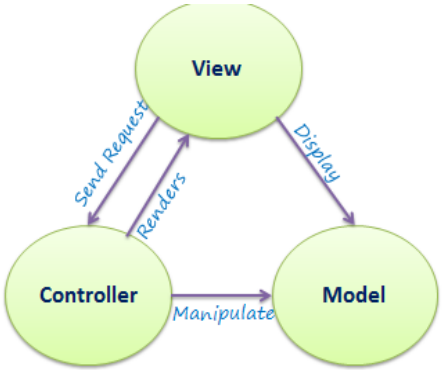

Figura 2: MVC

4.7 Diretrizes

Frontend

- A aplicação é modularizada, podendo assim ser aplicado o carregamento tardio (lazyload), onde, determinado módulo será carregado apenas sob demanda;
- Componentizar sempre que possível o código que foi desenvolvido;
- Tipar as variáveis utilizadas para modelo;
- Utilizar interface para representação dos modelos que serão enviados pelas APIs consumidas no projeto;
- Utilizar o conceito de formulários reativos para aceleração do desenvolvimento;
- Utilizar o conceito de Pipes para transformar exibições de valores na view;
- Utilizar o conceito de Diretivas para ativar comportamentos específicos que serão reutilizados;
- Aprofundar na utilização de Observables do RxJS para melhor controle de requisições;
- Utilizar guardas de rotas para controlar as permissões da aplicação;
- Abstrair serviços muito utilizados para reutilização;
- Não utilizar variáveis globais na aplicação.

Backend

- Utilizar os padrões DTO e VO para trafegar as representações dos objetos de entidades entre as camadas do projeto e aplicações;
- Declarar os endpoints das APIs da aplicação seguindo as boas práticas do RESTful;
- Utilizar Bean Validation para validação dos dados nos objetos de entradas dos endpoints;
- Implementar serializable para as classes DTO/VO e as que representam as entidades no projeto;
- Criar Bean Validation customizado caso a validação seja utilizada em mais pontos no projeto, evitando assim a repetição de código na aplicação;
- Utilizar padrão MAPPER para conversão de DTO/VO para entidades e vice e versa;
- Criar checked exceptions sempre que necessário;
- Implementar hash e equals para as entidades;
- Utilizar transação NotSupported na declaração da classe BO, para os métodos que devem utilizar transações, especifique qual será a transação na assinatura do método referido;
- Não deverá ser utilizado números mágicos nas classes do projeto, uma alternativa é a criação de uma constante com descrição precisa do que esse determinado número representa;
- Use delegação ao invés de herança, se possível;
- Tente manter os métodos pequenos, desacoplados e com alta coesão;

- Javadoc na assinatura do método;
- Os EJB devem ser instanciados com a anotação @EJB, enquanto outros componentes que são gerenciados pelo container mas não são EJBs devem ser instanciados com @Inject;
- As implementações de regras de negócio devem estar cobertas por testes de unidade.

4.8 Restrições e definições Tecnológicas ou de Negócio

- JDK 12u1;
- Node v10.x;
- NPM 6.x;
- Angular 7;
- Angular Material 8.0;
- Java EE 8;
- Apache 2.4;
- Wildlfy 17.0.1.Final;
- Firefox 5.x;
- Google Chrome 59;
- Opera 12;

4.9 Interoperabilidade

O acesso a serviços externos ou a exposição de serviços providos pelo próprio sistema são realizados através das APIs JAX-RS e/ou JAX-WS;

4.10 Segurança

A segurança na camada de aplicação é controlada através do gerenciador de acesso e identidade *Keycloack*. A porta de entrada da aplicação, os *endpoints rest*, são configurados para permitir o acesso somente para os usuários/perfis pre determinados nos requisitos do sistema.

4.11 Escalabilidade

O sistema é construído baseado nas especificações JavaEE, que por sua vez, garantem através dos fornecedores, uma forma fácil e segura de escalar a aplicação horizontalmente. Alguns desses fornecedores que implementam as especificações JavaEE são a *RedHat* (Jboss e Wildfly) e a *Oracle* (WebLogic).

4.12 Ferramentas

- IDE Backend – Eclipse Photom;

- Versionamento CLI – Git Bash;
- IDE Frontend – VsCode;
- Gerenciador de pacotes Frontend – NPM.

4.13 Outras Decisões Relevantes

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

Convenção Java:
https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Angular Style Guide:
https://angular.io/guide/styleguide

A qualidade do código deve ser constantemente verificada com auxílio do *SonarQube*, utilizando as métricas aplicadas com profile disponibilizado pelo Ministério.

5 Desenho Modular

5.1 Diagrama de Componentes

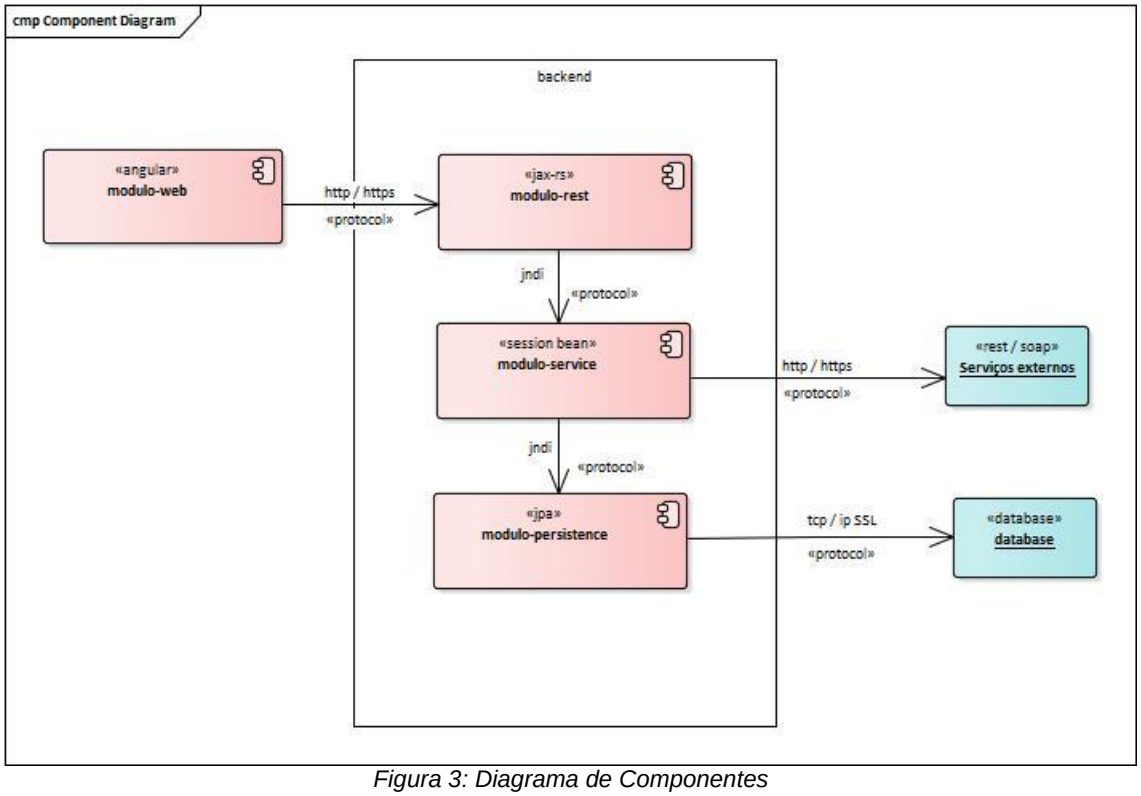

Figura 3: Diagrama de Componentes

6 Modelo de Implantação

6.1 **Diagrama de Implantação**

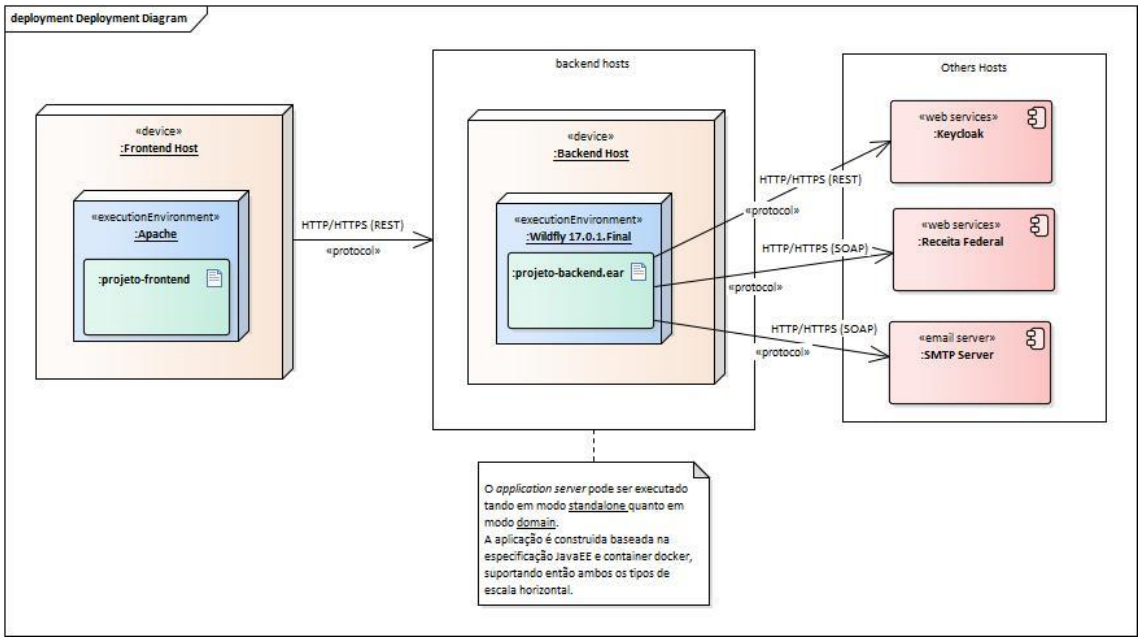

Figura 4: Diagrama de Implantação

7 Estrutura de Diretórios

Para os projetos Java deverão ser seguidas a estrutura de diretórios propostas pelo Maven, onde, cada diretório tem um proposito definido:

- **src/main/java**: Código fonte da aplicação;
- **src/main/resources**: Recursos que a aplicação necessita, como arquivo de propriedades com configurações, entre outros;
- **src/main/webapp**: Diretório para aplicações WEB;
- **src/test/java**: Código fonte com as classes com os testes unitários;
- **src/target/***: Classes compiladas e arquivos de distribuição do projeto, é gerado automaticamente quando o build da aplicação é solicitado.

Para a aplicação desenvolvida com Angular IO, a mesma é organizada com módulos que são divididos da seguinte forma:

- **Root module**: Contém as declarações de todos os componentes que serão usadas na aplicação, esse é o módulo que faz o *bootstrap* da aplicação;
- **Shared module**: Contém todos os componentes que são usados por outros módulos da aplicação, por exemplo: pipes e diretivas

- **Core module**: Contém todos os serviços da aplicação, esse módulo não contém componentes;
- **Feature modules**: Contém as implementações das funcionalidades da aplicação.

8 Revisões do Artefato

Data	Descrição	Responsável
23/06/2019	Versão inicial do documento.	Diogo Carvalho de Matos

9 Aprovação

Órgão/Setor	Nome	Data	Assinatura