STJ | Secretaria de Tecnologia da Informação e Comunicação

Arquitetura de Aplicações Web

- Guia de Referência -

Janeiro 2019

Versão 1.0.0

Anexo
(Art. 1º da Instrução Normativa STJ/GDG n. 4 de 15 de janeiro de 2019)

Índice

1 - Introdução

O objetivo deste documento é apresentar, de forma sucinta, a arquitetura padrão proposta para o desenvolvimento de novas aplicações web para o Superior Tribunal de Justiça.

2 - Objetivos

A arquitetura proposta foi concebida com os seguintes objetivos:

1. Simplicidade no desenvolvimento;
2. Compatibilidade com o que é utilizado no mercado;
3. Produtividade na implementação;
4. Flexibilidade para adaptação;

3 - Visão Geral

A arquitetura proposta para novas aplicações web tem como base a arquitetura de microsserviços. A aplicação é dividida em duas camadas principais: *front-end* e *back-end*. Essa divisão em camadas é ilustrada pela Figura 1.

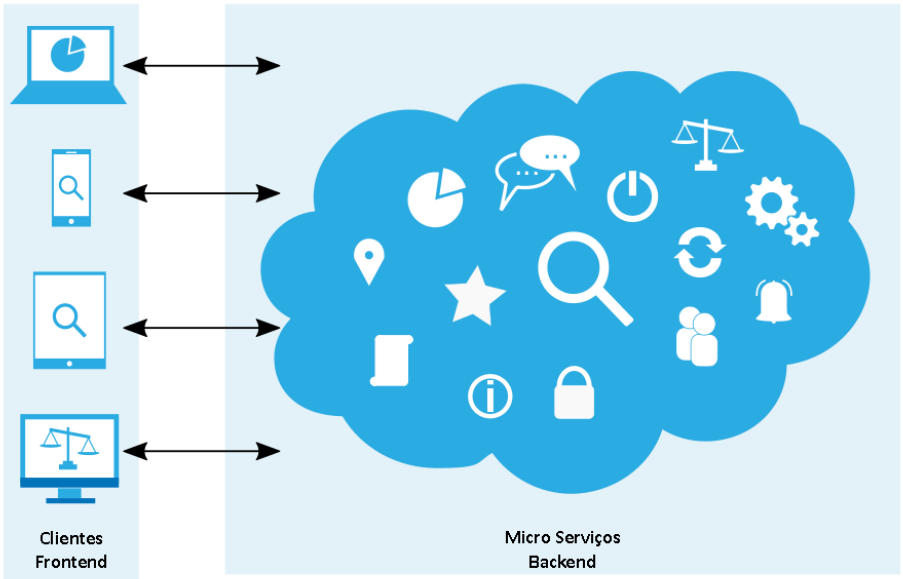


Figura 1 - Arquitetura de microsserviços

A parte *Back-end* é composta por microsserviços desenvolvidos com a tecnologia Java e o framework *open source* Spring. Já a camada *Front-end* é composta por componentes desenvolvidos com o framework Angular.

3.1. O que é a arquitetura de microsserviços?

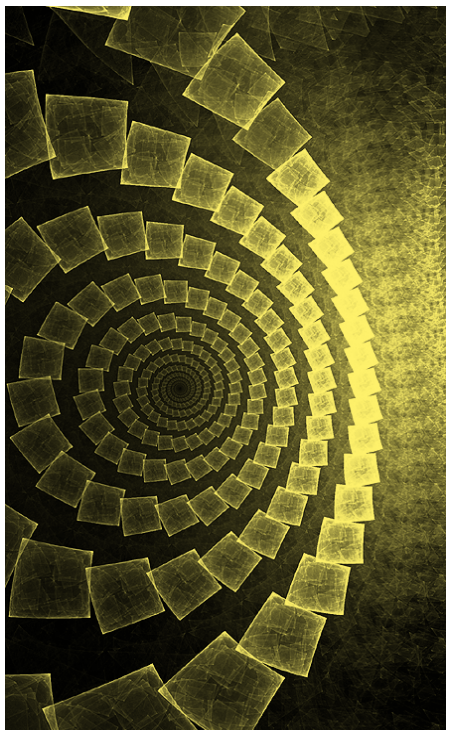
É um modelo de arquitetura de software baseado em um grupo de serviços separados, compartimentalizados, interdependentes e especializados em um domínio do negócio que se comunicam por uma camada a fim de formar uma aplicação.

3.2. Por que usar a arquitetura de microsserviços?

Na arquitetura de microsserviços, uma aplicação é dividida em uma suíte de pequenos serviços, cada um executando seu próprio processo e se comunicando através de mecanismos leves, muitas vezes por meio de uma API com recursos HTTP. Esses serviços são construídos em torno de áreas de negócios e funcionam através de mecanismos de implantação independentes e automatizados.

3.3. Padrões arquiteturais adotados

Os principais padrões aplicados na arquitetura proposta são:



Single responsibility principle

- Cada classe ou componente deve ter apenas uma responsabilidade.

API gateway

- Construir a aplicação como um conjunto de microsserviços;
- Decide como a aplicação cliente irá interagir com os microsserviços;
- Cada microsserviço expõe um conjunto de *end points*.

Service registry

- Determina a localização das instâncias do serviço para enviar as respectivas requisições;
- O serviço é incluído em um servidor de registros e pode ser identificado por meio de um outro serviço, o qual possui a prerrogativa de descoberta dos demais serviços.

Service discovery

- Descoberta de serviços;
- Os serviços são executados em ambiente virtualizado que podem mudar o local ou o número de instâncias de acordo com a demanda;
- Cada serviço é identificado unicamente e registrado para que seja descoberto pela aplicação cliente.

Requisitos de segurança

O desenvolvimento dos serviços deve observar as práticas de desenvolvimento seguro descritas no seguinte link:

- Padrão JWT Web Token (OAuth2) - http://intranet/confluence/pages/viewpage.action?pageId=10616867

4 - Representação Arquitetural dos Serviços

4.1. Framework Spring Cloud

O Spring Cloud é um framework composto por uma série de componentes, entre eles:

- **Spring Cloud Config Server**: é responsável por permitir que as aplicações mantenham suas configurações (propriedades e parâmetros) armazenadas num ponto central. Com configurações e arquivos de propriedades armazenados e versionados remotamente, facilita-se a reconfiguração das aplicações, pois não há a necessidade de acessar cada servidor da nuvem, onde elas estão instaladas, para executar essa tarefa;
- **Spring Cloud Netflix**: integra aplicações construídas com Spring Cloud a diversos componentes *opensource*, tais como Eureka (registrador e balanceador de serviços), Hystrix (biblioteca com algoritmos para construção de sistemas tolerantes a falhas) e Zuul (serviço de borda para roteamento, monitoramento e segurança);

- **Netflix Eureka**: O componente Netflix Eureka permite o registro dos serviços que estão conforme eles surgem no ambiente de execução;
- **Netflix Ribbon**: O componente Netflix Ribbon é usado pelos consumidores do serviço para localizar um serviço em tempo de execução. O Ribbon usa a informação disponível no Eureka para localizar as instâncias do serviço. Se mais de uma instância for encontrada, o Ribbon fará o devido balanceamento de carga. O Ribbon é integrado em cada serviço;
- **Netflix Zuul**: O Zuul é um "porteiro" para o mundo externo, o qual impede a execução de requisições não autorizadas. Ele também provê um ponto de entrada único e conhecido para o ambiente de microsserviços e usa o Ribbon para localizar as APIs disponíveis, efetuando o roteamento das requisições externas para a instância de serviço apropriada.

Componente da arquitetura	Spring, Netflix OSS
Centralização das configurações dos serviços	Spring Cloud Config Server
Registro e descoberta de serviços	Eureka
Servidor de fronteira	Zuul Server
Gerenciamento de configuração central	Spring Cloud Config Server
Roteamento dinâmico e balanceamento de carga	Ribbon
API's protegidas com OAuth 2.0	Spring Cloud + Spring Security OAuth2
Monitoramento	Hystrix Dashboard
Tolerância a falhas	Hystrix
Comunicação entre serviços	Feign

Tabela 1. Componentes da arquitetura

A Figura a seguir ilustra a disposição destes componentes de serviço na arquitetura proposta:

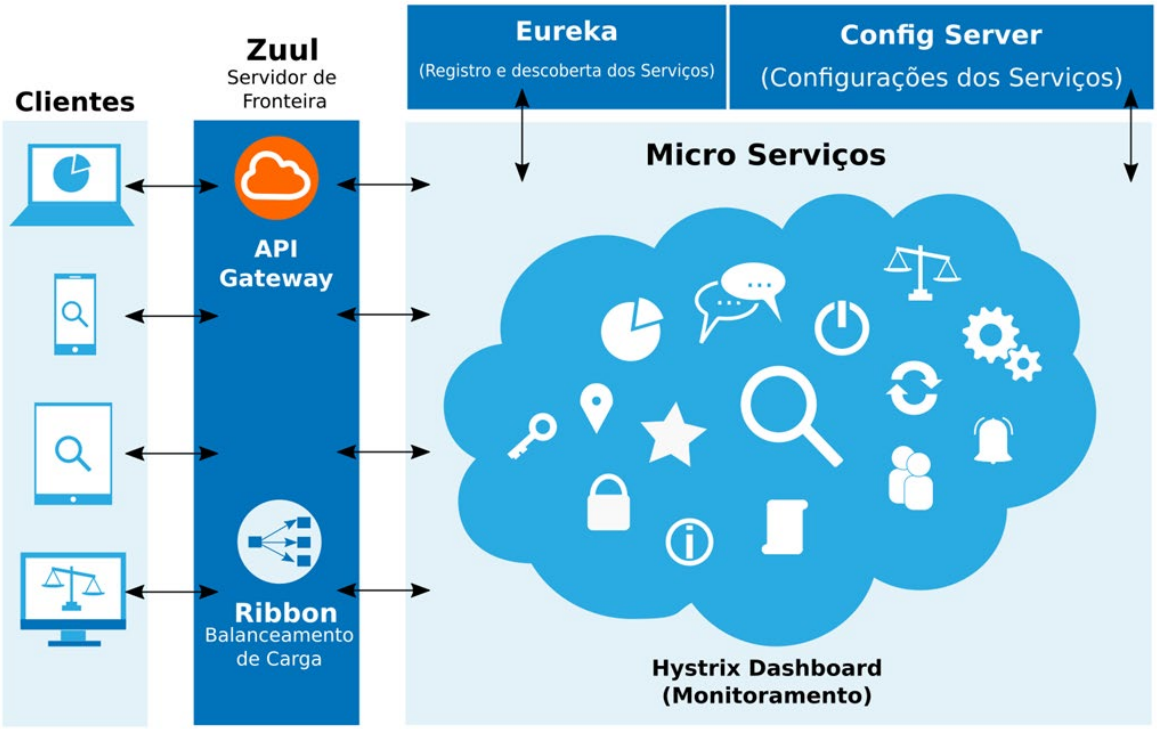


Figura 2. Distribuição dos componentes na arquitetura de microsserviços

5 - Visão de Componentes

5.1. Organização dos pacotes

Um pacote é um agrupamento lógico de classes e relações entre essas classes. O diagrama a seguir apresenta a organização em módulos lógicos e especificação de interfaces e dependências entre módulos.

O pacote **controller** agrupa as classes que respondem as requisições da interface. O controller chama os métodos que implementam a lógica de negócio localizados no pacote **service**. O service utiliza as interfaces contidas no pacote **repository** para realizar as consultas e acessar as entidades presentes no módulo **justiça-entity**. Para enviar os dados para a interface, a entidade é convertida por meio do **assembler** para um **DTO**, o qual possui apenas os dados necessários. A Figura a seguir demonstra essa distribuição e as comunicações.

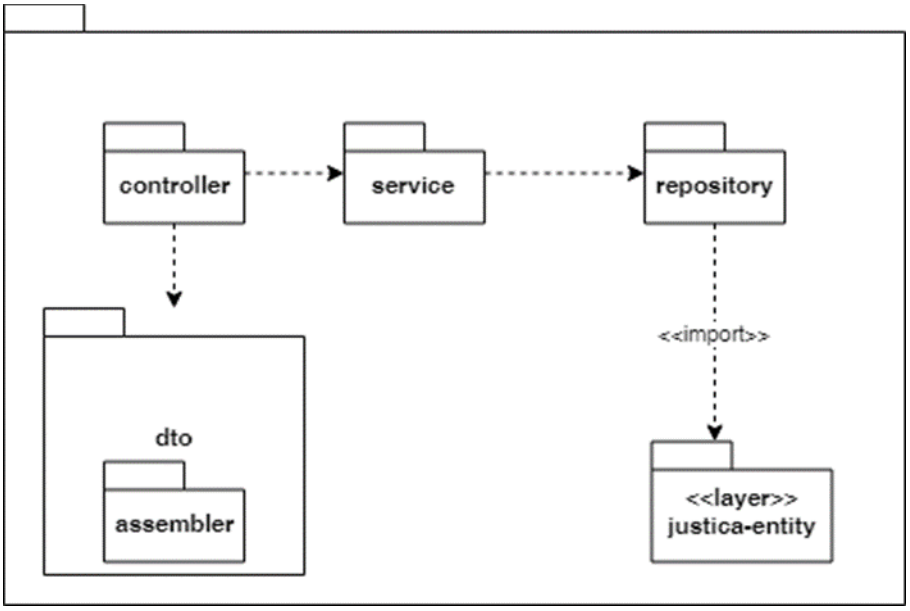


Figura 3. Estrutura dos pacotes

De maneira sucinta, as classes do projeto são divididas entre os pacotes da seguinte forma:

- **Config:** classes de configuração interna do projeto;
- **Controller:** classes que definem os *endpoints* e recebem as requisições HTTP oriundas de aplicações;
- **Service:** classes que implementam as regras de negócio e demais validações;
- **DTO:** classes que definem os objetos do tipo *Data Transfer Object*, os quais são utilizados para retornar os dados necessários à aplicação requisitante;
- **Assembler:** localizado no pacote **DTO**, agrupa as classes que convertem as entidades mapeadas do banco para os seus respectivos DTOs;
- **Repository:** interfaces que manipulam os dados de uma entidade em comunicação direta ao banco de dados.

5.2. Módulos que compõem cada projeto

Alguns projetos são fundamentais e base para a arquitetura de serviços do STJ, quais sejam:

- **Nome_do_projeto-entity:** concentra todo o mapeamento JPA das tabelas do banco de dados para as classes de entidade do Java;
- **Config-server:** possui as configurações necessárias para que os serviços que compõem um conjunto funcionem. Abriga um arquivo do tipo YML, o qual define configurações extras para cada novo microsserviço desenvolvido. Além disso, cada microsserviço possui arquivos desse mesmo tipo, que possuem configurações específicas para cada ambiente de execução;
- **Eureka-server:** permite o registro dos serviços no ambiente de execução;
- **Zuul-server:** funciona como um ponto de entrada único e conhecido para o ambiente de microsserviços.

5.3. Módulo *Front-end*

O Angular é um framework *opensource* para *front-end* mantido pelo Google, disponibilizado via NPM (gerenciador de pacotes) com repositório hospedado no *Github*. Frisa-se a diferença entre o *AngularJS*, *framework* distinto, e o *Angular*, aqui definido como padrão e sucintamente descrito. A seguir, demonstramos a estrutura básica de uma aplicação desenvolvida com a tecnologia Angular:

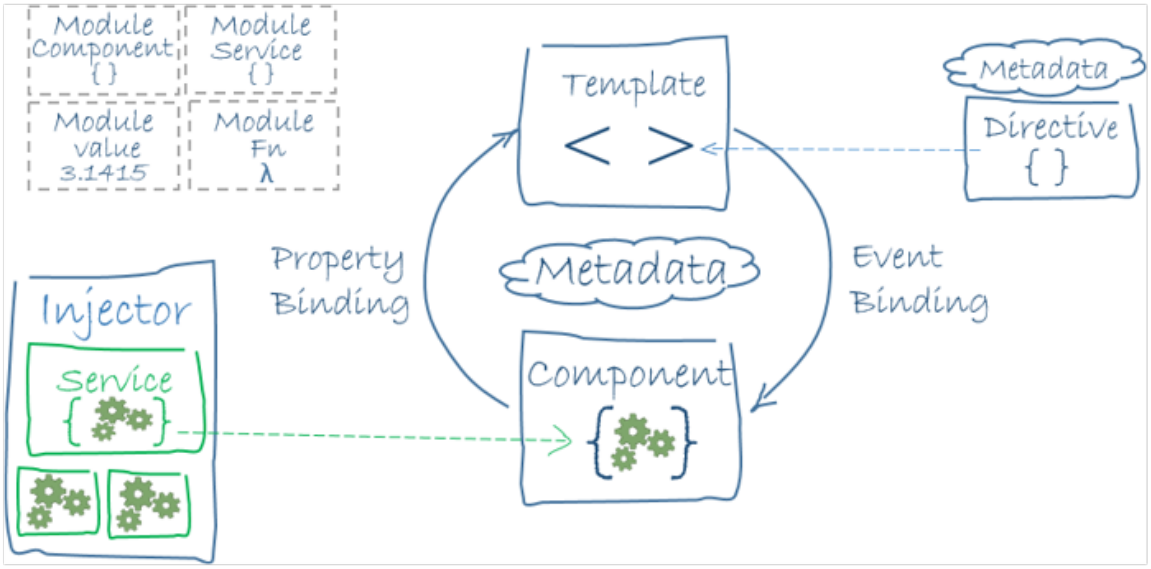


Figura 4 - Detalhamento da estrutura do Angular.
Fonte: https://angular.io/guide/architecture. Acessado em 03/10/2018.

- **Module:** classe que contém a declaração de todos os **Components**, **Services** e outros **Modules** necessários ao funcionamento de um domínio da aplicação;
- **Component**: classe que contém os dados da aplicação e a implementação das regras de negócio. É associada a um **Template**.;
- **Template**: classe HTML a qual define a parte da aplicação visível pelo usuário (*user interface* - UI) a partir dos dados existentes no **Component**;
- **Service**: classe com propósito bem definido, sem **Template** associado, portanto, responsável apenas por processamento de dados. Geralmente utiliza-se **Service** a fim de realizar a comunicação da aplicação Angular com serviços REST, estabelecer uma comunicação entre dois ou mais **Components**, centralizar algum processamento comum existente e fazer guarda de rotas.

6 - Visão de Implantação e Execução

A implantação dos projetos desenvolvidos na arquitetura proposta neste documento será realizada numa estrutura de contêineres (Docker). Esta tecnologia permite empacotar e isolar aplicações com todo o ambiente de tempo de execução e todos os arquivos necessários para executar os aplicativos. A Figura 5 apresenta a visão geral de aplicações container.

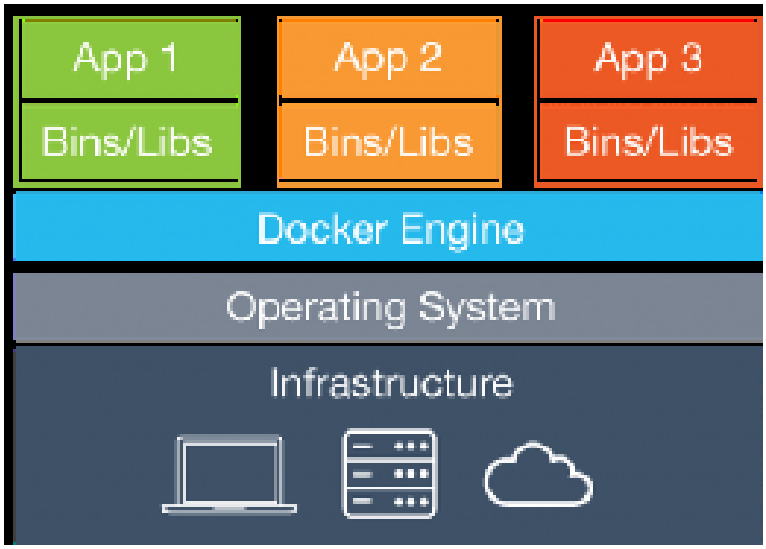


Figura 5. Contêiner

7 - Referências

PARK, Chanwook. *What are microservices?* http://microservices.io/index.html. Acesso em 01/02/2018.

PARK, Chanwook. *A pattern language for microservices.* http://microservices.io/patterns/index.html. Acesso em 01/02/2018.

THOUGHTWORKS. *Technology Radar Vol. 17: Nossas ideias sobre tecnologias e tendências que estão moldando o futuro.* https://www.thoughtworks.com/pt/radar. Acesso em 01/02/2018.

SPRING. *Spring by Pivotal. https://spring.io/*. Acesso em 01/02/2018.

MOZAFFARI Babak. *Reference Architectures 2017 Microservice Architecture.* https://access.redhat.com/documentation/en-us/reference_architectures/2017/pdf/microservice_architecture/Reference_Architectures-2017-Microservice_Architecture-en-US.pdf . Acesso em 01/02/2018.

LUKYANCHIKOV, Alexander. *Microservice Architectures With Spring Cloud and Docker.* https://dzone.com/articles/microservice-architecture-with-spring-cloud-and-do. Acesso em 01/02/2018.

Ambiente Docker STJ. http://colabora/sites/conhecimento/Lists/Procedimento/DispForm.aspx?ID=1331. Acesso em 27/07/2018.

CRUZ, Fábio. *Scrum e Agile em Projetos - Guia Completo: Conquiste sua certificação e aprenda a usar métodos ágeis no seu dia a dia.* Editora Brasport: Fev/2015.