# Desafio VR Desenvolvimento – Leandro Carvalho Camargos

1. Recebemos um código desenvolvido por terceiros de um sistema que possui volume alto de lógica de negócio e apresenta a seguinte característica:

O sistema recebe requisições Rest, está dividido em camadas e possui classes de domínio.

- O controller recebe a requisição e está com toda lógica de negócio. Monta e repassa o domínio para a aplicação;

- A aplicação tem a responsabilidade de repassar o objeto pronto para o repositório;

- O repositório apenas persiste os objetos mapeados do hibernate através de spring data;

- O domínio apenas faz o mapeamento para o BD;

- Nenhum teste unitário foi escrito.

- O sistema está escrito em java para rodar como spring boot.

Apresente observações/problemas sobre essa solução.

Comente qual a sua estratégia para melhorar este sistema em termos de qualidade e manutenção.

1. Descreva quais são as principais limitações ao se adotar servidores de aplicação em uma arquitetura orientada a microsserviços.
2. Quais são os principais desafios ao se adotar aplicações do tipo "Embedded Servlet” em relação a aplicações desenvolvidas para um application server?
3. Defina os resources e métodos http para o controller abaixo utilizando annotation do Spring MVC:

@RestController

public class PedidoResource {

public ResponseEntity<ServiceResponse<PedidoVO>> consultar(String idPedido) {

...

}

public ResponseEntity<ServicePageableResponse<List<PedidoVO>>> listar(PedidoListarDto pedidoListarDto) {

...

}

public ResponseEntity<ServiceResponse<Void>> criar(PedidoVO pedido) {

...

}

public ResponseEntity<ServiceResponse<Void>> alterar(

String idPedido,

PedidoVO pedido) {

...

}

public ResponseEntity<ServiceResponse<Void>> excluir(

String idPedido,

PedidoVO pedido) {

...

}

public ResponseEntity<ServiceResponse<Void>> atualizarStatus(

String idPedido,

Status status) {

...

}

public ResponseEntity<ServiceResponse<Void>> atualizarVencimentoPedido(

String idPedido,

VencimentoPedidoDto vencimentoPedidoDto) {

...

}

}

**Resposta Questão 1:**

A lógica de negócio não deveria estar na controller, e sim em uma camada de “Service” aonde as regras de negócio, e manipulação das informações deverá ocorrer, facilitando assim reaproveitamento de lógicas em comum entre os serviços e deixando de delegar funções extras ao que realmente a controller deve fazer.

Não é aconselhável expor na controller os objetos relacionais do Banco de dados, é aconselhável usarmos VO’s e DTO’s para fazermos esse transporte de informações entre as camadas, diminuído assim consideravelmente o acoplamento.

É fortemente aconselhável termos uma boa cobertura de teste unitário afim de garantirmos a qualidade do software, e evitar algum comportamento inesperado do sistema, e também termos a certeza que nenhum serviço vá responder algo diferente ao que ele foi criado.

E ainda, para garantir uma melhor qualidade e manutenção, a adoção de uma ferramenta de inspeção contínua da qualidade do código como SONARQUBE para garantir periodicamente se as métricas de qualidade de código bem como cobertura dos testes unitários estão satisfatórias. Bem como a adoção de uma ferramenta de automação de compilação como MAVEN para ajudar gerenciar as Libs externas utilizadas, bem como compilar o código.

**Resposta Questão 2:**

Ao se utilizar Servidores de Aplicação para micro serviço, perdemos um pouco da escalabilidade por serviço, muitas vezes tendo que escalar um cluster inteiro, com vários assuntos, apenas por que um deles está demandando mais recursos, dificulta a granularização das configurações que cada microserviço possa exigir, e tendo vários serviços sob um mesmo servidor temos sempre a possibilidade de um serviço em específico derrubar outros serviços no mesmo Servidor de Apliação.

Além disso os servidores de aplicação devem ser implantados além do microserviço. A configuração geralmente é bastante complexa, às vezes ainda mais complexa do que a configuração do próprio microserviço. Isso torna os scripts de automação mais difíceis de escrever.

E também é bastante oneroso manter os Servidores de Aplicação com as mesmas configurações em cada ambiente (Teste Integrado, QA, pre prod, prod, etc). Problema que se agrava um pouco mais ao adotarmos a entrega contínua, onde comumente teremos várias fases de e mm cada uma dessas etapas, possivelmente a aplicação será implantada em um ambiente diferente - e essa pipeline será executada várias vezes ao dia.

Já na parte de desenvolvimento, ter um servidor de aplicação significa uma etapa a mais no ciclo desenvolvimento – teste, pois vai precisar sempre compilar e compactar o microserviço em um WAR e posteriormente o Servidor de Aplicação vai fazer a descompactação subir totalmente o microserviço, para só assim possamos vê-lo rodando.

A implantação de um App Server para cada um desses microserviço adiciona uma sobrecarga considerável. Neste caso um número maior de instâncias será necessário comparado ao modelo de configuração tradicional. Além disso, os microserviços se concentram em infraestrutura especializada: um Servidor de aplicações pode ser uma boa escolha para o serviço no front-end. Mas para análise de dados, uma infraestrutura de Big Data como o Hadoop, será mais apropriado. Se houver algum tipo de processamento em lote, Spring Batch pode ser a melhor escolha.

Os Microserviços aumentam o número de componentes independentemente deployados. Isso significa que uma infraestrutura complexa deve ser evitada.

**Resposta Questão 3:**

Quando a empresa já possui App Servers rodando e ela já está acostumado com eles em produção. Mudar para embedded servers vai mudar uma boa parte de seus processos.

O suporte para deploy e monitoramento pode acabar tendo que ser feito com ferramentas genéricas às quais as operações podem não estar familiarizadas, como Puppet, Chef ou Ansible. Apesar de elas serem relativamente fáceis de configurar.

E ainda, **q**uando se tem várias aplicações em mesma máquina, infraestrutura de servidor ficaria repetida (aumentado espaço em disco e memória RAM) e poderia até causar conflitos de portas por exemplo.

O Tunning de servidores fica dificultado caso seja necessário.

Aplicação e servidor (às vezes banco de dados) são transportados juntos, o que aumenta o tamanho da aplicação.

Um grande desafio ao se usar microserviços com "Embedded Servlet” em relação a aplicações tradicionais desenvolvidas para application server é o controle de transações distribuídas, coisa que por exemplo o RMI geralmente trata nativamente, ao se usar microserviçoes, temos que optar por outros padrões para garantir essas transações, como SAGA Coreografia ou SAGA Orquestração por exemplo.

**Resposta Questão 4:**

@RestController

@RequestMapping("/pedido-service")

public class PedidoResource {

@GetMapping(value = "/{id}")

public ResponseEntity<ServiceResponse<PedidoVO>> consultar(@PathVariable("id") String idPedido) {

...

}

@RequestMapping(method= RequestMethod.GET, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServicePageableResponse<List<PedidoVO>>> listar(@RequestBody PedidoListarDto pedidoListarDto) {

...

}

@RequestMapping(method= RequestMethod.POST, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServiceResponse<Void>> criar(@RequestBody PedidoVO pedido) {

...

}

@RequestMapping(value = "/{id}", method= RequestMethod.PUT, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServiceResponse<Void>> alterar(@PathVariable("id") String idPedido, @RequestBody PedidoVO pedido) {

...

}

@RequestMapping(value = "/{id}", method= RequestMethod.DELETE, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServiceResponse<Void>> excluir(@PathVariable("id") String idPedido, @RequestBody PedidoVO pedido) {

...

}

@RequestMapping(value = "/status/{id}", method= RequestMethod.PUT, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServiceResponse<Void>> atualizarStatus(@PathVariable("id") String idPedido, @RequestBody StatusDTO status) {

...

}

@RequestMapping(value = "/vencimentopedido/{id}", method= RequestMethod.PUT, produces = MediaType.APPLICATION\_JSON\_VALUE, consumes = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<ServiceResponse<Void>> atualizarVencimentoPedido(@PathVariable("id") String idPedido, @RequestBody VencimentoPedidoDto vencimentoPedidoDto) {

...

}

}