# Desafio VR Desenvolvimento

1. Recebemos um código desenvolvido por terceiros de um sistema que possui alto volume de lógica de negócio e apresenta as seguintes características:

- 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(is) a(s) sua(s) estratégia(s) para melhorar este sistema em termos de qualidade e manutenção. Justifique suas decisões.

Pelo que entendi, é apenas uma classe de controle, creio que deve ser separada em mais de uma classe de controle e deva ser direcionada de forma mais específica para uma ou mais classes de domínio, de forma a melhorar e facilitar o entendimento do código.

Os repositórios podem sim serem usados para persistir os objetos mapeados, mas é interessante que estes objetos sejam verificados se podem ou não serem salvos no banco de dados antes, podendo implementar dentro do próprio modelo dos objetos a verificação se válido, ou ser implementado no repositório.

Também é interessante separar o repositório por classe de domínio, de forma a melhor organizar o código.

Eu particularmente costumo implementar as validações de objeto dentro do modelo de objeto, tipo, verificar se um campo pode ou não ser nulo, se está ou não dentro de um range de valores, se pode ou não conter a quantidade de caracteres. Isso para validar o objeto antes de salvar. Neste caso, a classe de domínio, ou modelo, neste caso implementaria esta validação, pois é algo específico do objeto.

Eu, particularmente, também separaria a requisição de dados da classe de controle, montando uma classe anotada como @RestController, ficando este tipo de classe de controle responsável apenas por receber as requisições, e a partir dessa classe chamaria a classe de controle com as regras de negócios. Isso pode aumentar o volume de código, a princípio, mas torna muito mais fácil o entendimento do fluxo dos dados dentro da aplicação.

Outra vantagem é que separando o controle dessa forma, caso seja necessário alguma manutenção no código, é possível executar os testes de uma maneira mais direcionada, onde realmente necessita da manutenção.

Dessa forma o código é refatorado, fica melhor escrito, melhor dividido, orientado de forma mais consistente entre cada uma das classes de domínio, controladores específicos para as necessidades de cada classe de domínio, ou conjunto das mesmas, repositórios específicos para cada classe de domínio. Constituindo assim um modelo MVC melhor arquitetado.

Também há necessidade de escrever os testes unitários necessários, mas se o código já estiver funcionando, é interessante escrever os testes unitários a medida que for sendo necessário, ou seja, a medida que o código for sendo retatorado. Se esta aplicação foi entregue, ainda sem estar em funcionamento, ai digo que seja necessário escrever e executar os testes unitários para as operações da aplicação antes de lança-la.

1. Descreva quais são as principais limitações ao se adotar servidores de aplicação em uma arquitetura orientada a microsserviços.
2. Complexidade presente nos sistemas distribuídos.
3. Em testes de microserviços, além do teste do próprio microserviço, é necessário também que todos os serviços que este depende estejam disponíveis.
4. Maior complexidade em gerenciar e manter a consistência de múltiplos bancos de dados.
5. Casos de uso que utilizam de vários serviços podem requerer atualizações em vários serviços dependentes e alinhamento com vários times de desenvolvimento (isso deve ser evitado).
6. A rede deve ser confiável.
7. Redução do número de requisições utilizando recurso de api-gateway, reduzindo também a latência.
8. Certificar que as operação são thread-safe.
9. Atualmente, diversas aplicações escritas em Java estão deixando de serem desenvolvidas para rodarem em servidores (JBoss, Tomcat), adotando ferramentas que disponibilizam um servidor embutido na própria ferramenta. Quais são os principais desafios ao se tomar uma decisão dessas? Justifique sua resposta.

Considerando um cenário de micro serviços, soluções usando o modelo embarcado agiliza e deixa o ciclo de desenvolvimento mais rápido, uma vez que não é necessário o tunning de serviço.

Contudo existe vantagens e desvantagens no modelo de servidores embarcados x servidores externos.

Entre as vantagens estão um ambiente de produção e desenvolvimento mais parecidos, o que diminui erros na aplicação. Uma melhor distribuição, pois o aplicativo roda mais facilmente em outras máquinas, pois não é necessário gastar tanto tempo com configuração do ambiente de produção. E as aplicações se tornam independentes entre si. Caso aconteça alguma coisa com uma aplicação, não afeta as outras aplicações dentro do servidor.

Mas existe algumas desvantagens, como por exemplo: aplicação e servidor sendo transportados juntos pode aumentar consideravelmente o tamanho da aplicação, principalmente se o banco de dados for levado junto com a aplicação. Se houver mais de uma aplicação na mesma máquina, a infraestrutura pode ficar repetida, outra vez, aumentando o espaço em disco e memória RAM necessária para execução, além de conflitos de portas lógicas, durante as comunicações. E caso seja necessário o tunning de servidores, pode tornar bem mais complexo estas configurações.

Por isso devemos certificar que a configuração das máquinas, em questão de memória e espaço em disco estejam de acordo com seu propósito, além disso, ter em mente que a comunicação entre as aplicações sejam escaladas e orientadas corretamente para não executarem processos simultaneamente utilizando as mesmas portas, ou que a utilização das portas não seja exclusiva, mas permita o uso compartilhado entre as aplicações.

1. Teste prático (em anexo)