Desafio VR Desenvolvimento

* 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.

-A controller não deve conter regras de negócio, visto que sua responsabilidade é ser a porta de entrada da aplicação, e não executar o processamento e as regras de negócio. Além de violar o principio de responsabilidade única, o que deixa o código mais difícil de fazer manutenção, fazer de tal forma aumenta o acoplamento das regras de negócio à tecnologia utilizada, então se um dia por exemplo, for especificado que o REST será substituido por outra tecnologia como SOAP ou até mesmo o consumo de uma fila ou tópico, será mais dificil mudar este código.

-Se não houvessem regras de negócio a serem implementadas na camada de service, não seria incomum o service apenas invocar o BD através da camada de repository. Porém é possível executar o mapeamento através do próprio Model do objeto a ser guardado utilizando anotações como @Sql, portanto há aí uma oportunidade de melhoria para deixar o código mais limpo.

-Testes unitários são muito importantes para a robustez e facilidade de manutenção da aplicação! Um código sem testes é um código sem garantia de funcionamento.

-No geral as estratégias para melhorar o sistema seriam:

-Tirar todas as regras de negócio da controller e colocar no service, deixando no máximo alguma validação dos dados de entrada ou talvez alguma camada de autenticação para garantir que a chamada é valida;

-Executar mapeamento para o banco de dados no próprio model a ser guardado através de anotações com o nome da tabela e de cada coluna, deixando assim no service apenas a montagem do objeto a partir do DTO recebido. Nesse ponto seria importante decidir também se seria necessário uma distinção entre o Model em que as regras de negócio são aplicadas, e a entity que é o que efetivamente vai ser guardado no BD.

-Escrever testes unitários para garantir que cada uma das regras de negócio estão sendo executadas corretamente, afim de guiar o processo de desenvolvimento e facilitar manutenções futuras.

* Descreva quais são as principais limitações ao se adotar servidores de aplicação em uma arquitetura orientada a microsserviços.

Quando tratamos de microsserviços, estamos pensando em projetar sistemas com responsabilidades bem definidas conectados entre si para executar alguma função. E estes mesmos sistemas devem estar preparados para lidar com mudanças na quantidade de requisições que estes recebem, o que pode gerar dificuldades para servidores on-premise, já que um aumento de requisições vem junto de um aumento de consumo de recursos computacionais, que dentro de um ecossistema de microsserviços pode tornar cada passo do processo muito lento, prejudicando a experiência do usuário, ou até mesmo causar uma falha total, prejudicando a imagem de quem mantem aquele sistema frente a seus clientes.

Por este motivo hoje em dia quando falamos de microsserviços, quase sempre falamos também de utilizar servidores Cloud, que apesar de no fundo também serem servidores, possuem uma capacidade total de recursos muito vasta e mecanismos para aumentar ou diminuir a quantidade de recursos utilizados dependendo da demanda, diminuindo a chance de causar alguma indisponibilidade e mantendo os custos adaptáveis dependendo do que for utilizado.

* 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.

Digamos que nossa aplicação não esteja em uma arquitetura de microsserviços. Ou seja, temos diversos serviços que não são isolados e contidos em si, com algum nível de dependencia de uns com os outros. Nesse caso, pode ser mais conveniente ter todos estes serviços rodando em um único servidor Tomcat ou Jboss por exemplo, do que ter cada aplicação com seu próprio servidor, pois você teria que gerenciar diferentes portas, propriedades e no geral gerenciar e monitorar cada um deles, o que poderia ser mais fácil caso fosse um único servidor.

Porém, uma vantagem muito grande de utilizar servidores embutidos na aplicação, é que a queda no servidor de uma aplicação não afeta a outra, possibilitando utilizar padrões como circuit breakers para evitar que a queda de um serviço tenha um impacto muito grande na experiência do usuário final. Além do mais, existem diversas formas de lidar com muitos servidores ao mesmo tempo, como ferramentas de gerenciamento como o spring admin, api gateways como o zuul para lidar com diversas aplicações usando diversas portas diferentes, entre outros.

Portanto, se a aplicação em que estamos considerando usar servidor embutido ou externo exibir um nível alto de acomplamento, talvez até mesmo um monolito, pode valer a pena usar um único servidor externo, já que se houver um problema em um serviço provavelmente já ocorrerão problema em outros, então assim fica melhor para facilitar o gerenciamento do servidor. Porém se estamos falando de microsserviços, ter servidores embutidos é mais prático e fácil, além de não gerar acomplamentos no nível do servidor.

* Teste prático (em anexo)